Global 16-bit Development System
Toolkit Manual
Version 8.1

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or
transmitted, in any form or by any means,
electrical, mechanical, photocopying,
recording or otherwise, without
the prior permission of
TIS Software Limited.

Copyright 1994 -2001 Global Software

MS-DOS is a registered trademark of Microsoft, Inc.

Windows NT is a registered trademark of Microsoft, Inc.

Unix is a registered trademark of AT & T.

C-ISAM is a registered trademark of Informix Software Inc.

D-ISAM is a registered trademark of Byte Designs Inc.

Btrieve is a registered trademark of Pervasive Technologies, Inc.

TABLE OF CONTENTS

Section Description Page Number

1. Introduction ... **???**

2. Command Programs ... **???**
2.1 $COPLB - Build Copy Library Index ... ???
2.2 $DEMO - Demonstration Command ... ???
2.3 $DEV - Program Development Dialogue Generator ... ???
2.4 $ED - Full Screen Editor ... ???
2.5 $FCOMP - File Comparison Utility ... ???
2.6 $JOURNAL - Journal of Screen Dialogue ... ???
2.7 $PATCH - File Patch Utility ... ???
2.8 $PDUMP - Formatted Program Dump Utility ... ???
2.9 $RECOVER - File Recovery Utility ... ???
2.10 ... $SCOMP - Source File Comparison Utility ?
2.11 ... $STACK - Stack Maintenance ?
2.12 ... $URMAIN - User System Request Data Library Maintenance ?
2.13 ... $VOL - List Volume Directory Utility ?
2.14 ... $XBUILD and $XPRINT - Global Cross-Reference Generators ?

3. Product Translator ... **???**
3.1 Introduction ... ???
3.2 Creating a Dictionary ... ???
3.3 Translation ... ???
3.4 Text Display ... ???
3.5 Check Dictionary ... ???
3.6 Error Messages and Stop Codes ... ???

4. Intermediate Code Language ... **???**
4.1 Foreword ... ???
4.2 The Intermediate Code 'Computer' ... ???
4.3 Byte String Instructions ... ???
4.4 Numeric String Instructions ... ???
4.5 Arithmetic Instructions ... ???
4.6 Transfer of Control Instructions ... ???
4.7 Status Switching Instructions ... ???
4.8 Parameter Stack Instructions ... ???
4.9 Register Load Instructions ... ???
4.10 ... Special Purpose Instructions ?

5. Metajob Management ... **???**
5.1 Foreword ... ???
5.2 The Metajob Description ... ???
5.3 The Metajob File Builder ($MJOB) ... ???
5.4 Using $MTEST to Initiate a Metajob Run ... ???
5.5 Writing your own Metajob Initiator ... ???

6. Macro Pre-Processor Language ... **???**
6.1 Introduction ... ???
6.2 Running $MACRO ... ???
6.3 The Preprocessor Language ... ???
6.4 $MACRO Options ... ???

APPENDICES

Appendix | Description | Pa

A Example Metajob System.. ???

B Error and Warning Messages from $MJOB......................... ???

C Error Messages from $MRUN..................................... ???

D Example $MACRO Listings.. ???

E $MACRO Error and Warning messages.............................. ???

1. Introduction

The Global Toolkit consists of a number of command programs designed to aid the production of large systems written in Global Cobol. Most of the commands are documented in alphabetical order following this introduction, but there are also five separate sections at the end. The first describes Product Translator, a facility that allows the user to alter the text contained in a program (e.g. error messages or prompts) without the need to edit, compile and link. The second contains a full account of the intermediate code language generated by $COBOL, while the last two deal respectively with the Macro Preprocessing language for use with Global Cobol, and with the Metajob Management language, which is a high level language for generating job control dialogue.

Most of the commands in the Global Toolkit are for use during program development, but some have wider uses. The $DEMO command allows a number of screens on a multi-user machine to be slaved from a single screen used by a demonstrator, so that anything displayed on the demonstrator's screen is echoed on all the other screens.

The $RECOVER program recovers files which have become corrupted due to I/O or software errors, and may also be used to 'undelete' files which have accidentally been deleted. The $PATCH program allows you to patch files and to create ZAPs to patch errors in programs. This utility is also very useful during program development to correct errors in programs in order to allow testing to continue.

For anyone developing a large system, the $XBUILD and $XPRINT commands will produce a single cross-reference listing of a large number of programs. The $VOL utility produces a detailed directory listing of a volume, which gives the contents of all the libraries. The $MACRO command interprets the Macro Preprocessing Language, which can be used to enable several slightly different versions of a program to be produced from a single source. The Metajob Management Language is an easy way to write complex, flexible jobs. For example, it can be used to install software onto a wide range of different configurations.

Programmers are likely to find the $DEV and $ED commands very useful. The $DEV command generates jobs to compile and link programs, and can greatly reduce the keying required as well as allowing a number of programs to be compiled in succession without operator intervention. The $ED command is a full screen editor, which can be used as an alternative to $EDIT.

1.1 Installation

The Global Toolkit is distributed on either one or two diskettes. Global Cobol must be installed prior to installing the Toolkit. The installation job puts the Toolkit commands on a new diskette or subvolume, SYSKIT, which will be formatted or allocated if required.

To install the Toolkit use the 'Install Global Software' entry in the menu, or run command $INSOFT directly. If you are installing onto diskette you must install onto one of the same format as SYSRES and with a capacity of at least 300K. If you are installing onto hard disk you can install onto any unit: the default is SYSKIT, if one is already allocated, or the first available subunit otherwise. When installing onto hard disk you will be asked how many operators will use $DEV: this is so that space can be left for the files created by

$DEV to hold defaults for each operator (2.5K per operator per partition).

The Toolkit commands listed below are distributed without serial numbers, so that you can give them to other users if you want. You must, however, copy the associated overlay programs and files at the same time. The unserialised commands are to be found in library P.TK.

Command	Associated Programs
$DEMO	$DEMEXEC
$FCOMP	None
$FIND	None
$JOURNAL	$JINTER
$MRUN	None
$MTEST	None
$PDUMP	None
$RECOVER	None
$STACK	None
$URMAIN	User system request data library
$VOL	None

Figure 1 - 'Free' Toolkit Commands and their Associated Programs

2. Command Programs

2.1 $COPLB - Build Copy Library Index

The $COPLB command is used to speed up the processing of copy libraries by $COBOL, $FORM and $XREF (or any other program using subroutine LIBR$). It does this by creating an extra index record at the end of a copy library, after the end of the file, so that LIBR$ need not scan the whole file to find the names and locations of the copy books it contains. This typically saves 3-10 seconds per compilation.

To create a copy library with an index you must run $COPLB and specify the input copy library and the new library to be created. For example:

```
GSM READY:$COPLB
$308 INPUT COPY LIBRARY:XY UNIT:205
$308 OUTPUT COPY LIBRARY:<CR> UNIT:206
```

The defaults for the output file if <CR> is keyed are the same name and unit as the input. If the output file has the same name and unit as the input then the original file is overwritten (but space must be available on the unit to create a copy of the input file).

Note that if the indexed copy library is subsequently edited, the index record will be lost and will have to be recreated by running $COPLB again.

It is recommended that you build indexes on all copy libraries for a program suite before a final packaging so as to decrease the compilation time required.

2.2 $DEMO - Demonstration Command

The demonstration command, $DEMO, allows some or all of the screens on the same computer to be slaved from a single screen, so that any displays on that screen appear on all screens simultaneously. It is intended for use when demonstrating Global software to more people than can usefully sit round one screen.

All the screens being slaved from the master screen must have the same terminal type as the master. Screens that are not the same type will produce garbled displays, and may in some cases cause the system to fail. You can only slave screens that have been signed on to the system by valid replies to the operator-id and terminal type prompts.

Any job which is currently executing on a slaved screen will continue as if it were running in background.

2.2.1 Starting a Demonstration

To start a demonstration, run command $DEMO. For each signed-on screen you will be prompted whether you wish to slave it from your terminal, for example:

```
$300 SLAVE SCREEN 2 OPERATOR ABC?:
```

Reply <CR> or Y to slave this screen, or N if you do not want to slave the screen if, for example, it is of a different terminal type. You can also reply <CTRL B> to slave this screen and all further screens or <CTRL A> if you do not want to slave this or any further screens.

After you have replied to all the prompts, your screen and all the slaved screens will be cleared. The available user area will be reduced by about 1100 bytes. You can now run any Global software you want to demonstrate, and any displays on your screen will be echoed on all the slaved screens.

2.2.2 Ending a Demonstration

To end a demonstration session, run the $DEMO command again. All the slaved screens will be cleared, and the system will be returned to normal.

2.3 $DEV - Program Development Dialogue Generator

The $DEV command generates dialogue to compile, link and cross-reference a program or series of programs. The command 'remembers' the last program processed for each operator, together with the associated units, copy libraries and subroutines used, so that if you want to recompile and relink the program you last compiled you need only key a few characters. Each partition is treated as a separate operator, with separately remembered defaults. You can also use $DEV to set up a long series of compilations and links which will execute without operator intervention, so that you can leave the machine unattended.

The command also allows special purpose job dialogue to be entered (to run a test, for example), and for messages and pauses to be inserted as required. It can be used in conjunction with specially written jobs to link a set of overlayed programs, or to macro preprocess source files.

The command operates by asking you to specify the type of dialogue required, for example a compilation, and then stepping through the dialogue for that command, offering you defaults for all the prompts. You can then either generate further dialogue, or cause the specified dialogue to be executed. You reply to any prompts in the dialogue in the usual manner: in particular you key any control responses as the normal keystrokes. <CTRL A>, for example, is not keyed as 8 separate characters.

2.3.1 The Dialogue

When you run $DEV it first displays its dialogue prompt:

```
$506 DIALOGUE:
```

The valid replies are given in the following table. They can be listed on a help screen by keying <CR> to the dialogue prompt.

Certain of the single character responses can be combined. The C (compile) dialogue can be followed by X, L, R, XL or XR. The C, J, M and T dialogue can be preceded by P. Thus, for example, CL is compile and link and PCXL is print source, compile, cross-reference and link. Note the CLX is not a valid combination: you must specify CXL.

Responses in the form L-xxxx execute specially named jobs that you have written to linkage-edit overlay structures. Similarly responses in the form M-xxxx execute jobs to macro preprocess a text file into several compiler input source files. The writing of such jobs is described later.

When you have specified all the required dialogue, reply E or Q to the dialogue prompt to start execution. The Q (Quiet) reply causes any

messages output by the executed programs to be suppressed and not displayed on the screen. It should be used if you want the execution to proceed in background, to prevent it being held up attempting to display messages.

Dialogue	Meaning
A	Abandon dialogue generation
C	Run $COBOL to compile a program
E	Execute dialogue
J	Run $JOB to create a job file
L	Run $LINK to linkage-edit a program
L-xxxx	Run Link Job $-L-xxxx to perform special linkage edit
M	Run $MJOB to compile a metajob
M-xxxx	Run Macro Job $-M-xxxx to run $MACRO to create several versions of source
P	Run $PRINT to print out a text file
Q	Execute dialogue in Quiet mode (no displays) so that job can proceed in background
R	Create Relocatable program (run $LINK twice, then $RELOC, then delete intermediate files)
T	Run $TAP to create a Terminal Attribute Program
X	Run $XREF to produce a cross-reference
:reply	Put this reply in dialogue (e.g. :$F)
+message	Display message (e.g. +NOW COMPILING SA110)
-pause	Display pause prompt (e.g. -MOUNT SA2 ON 230)

Figure 2 - $DEV Instruction Codes

If you terminate your reply to the dialogue prompt with <CTRL B> then the default responses for the selected dialogue will be used without you being prompted. Thus, for example, to repeat the compilation and link you last executed, you simply key:

```
GSM READY:$DEV
$506 DIALOGUE:CL<CTRL B>
.
.     dialogue for $COBOL and $LINK is displayed
.
$506 DIALOGUE:E
.
.      dialogue is executed
.
$506 DEVELOPMENT COMPLETE
GSM READY:
```

If you terminate your reply to the dialogue prompt with <ESCAPE> this allows you to execute the specified dialogue for every suitable file on the input unit. You will be prompted with the name of each file in turn: you must reply Y to process it, N or <CR> to ignore it, <CTRL A> to ignore all further files, <CTRL B> to process all remaining files or <CTRL C> to go back to the previous file. A reply of B is equivalent to a reply of Y followed by <CTRL B>, and is used to accept the defaults for the file selected.

2.3.2 Prompts within the Selected Dialogue

Once you have selected the dialogue, you will be presented with a series of prompts, with defaults displayed in parentheses. For example:

```
$506 DIALOGUE:C
      GSM READY:$COBOL
      $43 SOURCE (SA100):
```

Key <CR> to accept the default, or key a different value. The following special replies are also recognised:

<CTRL B> or terminating a reply with <CTRL B>, causes the default value to be used for all remaining prompts;

<CTRL C> deletes a prompt. This is only valid for those prompts which are repeated until <CR> is keyed, such as the copy library and option prompts in $COBOL. If the current default is <CR>, a reply of <CTRL C> has the same effect as <CR>;

<ESCAPE> cancels the current dialogue (since the last dialogue or file selection prompt) and returns to the dialogue prompt or file selection prompt.

Note that <CTRL A> has no special meaning, but is treated as part of the dialogue. In particular, you can reply <CTRL A> to a listing prompt to suppress the listing.

Many of the defaults are common to several sets of dialogue, and changing one changes the rest. For example, if you change the source file to SA110 this automatically changes the default first input file for $LINK to SA110 as well. Similarly, there is a single listing file unit. Thus, suppose you wish to compile and link program SA110, which is not the program you last compiled but does use the same units, copy libraries and subroutines as before. Then you need only key:

```
GSM READY:$DEV
$506 DIALOGUE:CL
      GSM READY:$COBOL
      $43 SOURCE FILE(SA100):SA110<CTRL B>
.
.      (remainder of dialogue is displayed)
.
$506 DIALOGUE:E
.
.      (dialogue is executed)
.
$506 DEVELOPMENT COMPLETE
GSM READY:
```

The defaults to be used when $DEV is next run are only updated when E or Q is keyed to the dialogue prompt: if you key A (abandon) they will be unchanged.

2.3.3 Overlay Linking Jobs

The L dialogue in $DEV can only be used with non-overlayed programs. If you need to link a set of overlayed programs you must write a special job, with a name of the form $-L-xxxx, where xxxx is any four characters. Such jobs must be placed on the volume containing $DEV, and may be put into a library P.DEVJOB if required. The job must accept three unit-addresses as parameters. These are, in order, the

compilation unit, the program unit and the listing unit. The job is run by keying L-xxxx to the dialogue prompt.

For small systems, with up to 10 programs in the overlay structure, you are recommended to relink the whole suite of programs each time. This means that only one link job is needed, and there is no danger of inconsistent versions. For larger systems, it may be desirable to provide several link jobs, each linking a group of related programs, together with a master job which calls all the jobs and links the whole system.

2.3.4 Macro Preprocessing Jobs

If you use $MACRO to preprocess source files then you can write jobs with names of the form $-M-xxxx, which can be executed from $DEV to run $MACRO with the required parameters. Such jobs must be placed on the same volume as $DEV, and may be put into a library P.DEVJOB if required. The job must accept two unit addresses as parameters. These are, in order, the macro source unit and the output source unit. The job, when run, should create all possible versions of the file.

2.3.5 Operating Notes

$DEV imposes certain limits on the dialogue generated, as follows:

- a maximum of 9 files can be specified for a link using the L dialogue;
- a maximum of 9 options can be specified in the C, X and L dialogue;
- a maximum of 9 copy libraries can be specified for C or X dialogue.

Only the first of these is ever likely to be a problem, and it can usually be avoided by using compilation libraries.

There is also a limit on the total dialogue generated, but this is very generous. For example, you can specify at least 20 executions of the CXL dialogue at the same time. You should however note that since $DEV creates a job (using JOB$) to execute the dialogue, this will reduce the available user area, and will degrade the performance of $COBOL if too many compilations are specified at once on a machine with a small user area.

$DEV allocates a 5.3K file on the unit containing $DEV for each operator who uses it, with separate files for each partition. The name of the file created is $$Doooop where oooo is the operator-id and p is the partition number.

2.4 $ED - Full Screen Editor

The $ED command is a full screen editor for amending text files. That is, it acts like a word processor in that you move the cursor to the text to be altered, and then insert, delete or change characters. However, it does not automatically wrap words onto the next line when you extend lines, since it is designed for editing program sources rather than documents.

$ED can be used as an alternative to the standard editor, $EDIT. In suitable circumstances, it is easier and faster to use. However, it requires much more processing time than $EDIT, and relies on a fast

response. Therefore, it is not recommended on multi-user machines when more than one other partition is active.

The principle limitation of the current version of $ED is in the handling of large files. $ED operates by first reading as much as possible of the file into memory, typically 250-1000 lines. If the file is larger than this, you must complete edits on the first portion and then read in the next. It is not possible to move text between such portions of the file using $ED, or to move text between different files.

There is a help file supplied which explains all the commands and facilities of $ED, and acts as a training aid. You should teach yourself about $ED by going through this file, editing it as instructed (these edits will have no permanent effect on the file, as it is treated specially by $ED). To start editing the help file, simply run $ED and key <CTRL C>.

2.5 $FCOMP - File Comparison Utility

The file comparison utility does a byte by byte comparison of two files and prints out any bytes which differ. The files may be of any type. The listing produced consists of a header page, giving any discrepancies between the file labels, and then pages consisting of the bytes that differ, one to a line.

2.5.1 Comparing Files

When you run $FCOMP, it prompts you for the names and units of the two files to be compared, for example:

```
$211 FILE 1:SADATA1  UNIT:202
$211 FILE 2:SADATA2  UNIT:202
```

You will then be prompted for the output unit on which the comparison report is to be produced:

```
$211 LISTING UNIT:
```

If you reply <CR>, it will be written to unit $PR; if you reply <CTRL A> it will be displayed on the screen. The comparison report is then produced, and when it is complete the file prompt is redisplayed, to allow you to compare further files:

```
$211 LISTING UNIT:<CR>
$211 COMPARE COMPLETE
$211 FILE 1:
```

Reply <ESCAPE> to quit.

2.6 $JOURNAL - Journal of Screen Dialogue

The $JOURNAL command, when activated, writes all subsequent screen dialogue to a print file so that it can be examined later. It can be used either to produce a hard copy listing of the dialogue, for example as a documentation aid or to help debug a program. It is particularly useful for debugging programs which, due to an error, are displaying non-ASCII characters on the screen. It can also be used to obtain hard-copy evidence of program errors for later investigation.

2.6.1 Activating the Journal

When you first run $JOURNAL it prompts you for the name of the journal file to be produced and for any special options required, for example:

```
$173 JOURNAL FILE:PRINT UNIT:203
$173 OPTION:
```

The default, if you key <CR>, is to produce file J.JNL on unit $PR.

Various options are available, and can be listed by keying <CTRL C>. Normally, however, the defaults will be satisfactory, and you can just key <CR> to the option prompt. The journal will then be activated, and a message giving the reduced high address will be displayed, for example:

```
$173 JOURNAL ACTIVATED - HIGH ADDRESS REDUCED FROM AB8C TO 9A56
```

The journal, when activated, occupies slightly over 4K of user area.

2.6.2 Deactivating the Journal

When you have finished using the journal, simply run $JOURNAL again to deactivate it:

```
GSM READY:$JOURNAL
GSM READY:
```

2.7 $PATCH - File Patch Utility

The file patch utility, $PATCH, will patch any type of Global file. It can also, if required, produce a ZAP of the patch, which can be applied to other similar files using $ZAP.

To prevent users from making changes to programs you have supplied (which could cause errors) $PATCH will not operate on files which are serial number protected. You can, however, still patch your own, unserialized, versions of these files.

The dialogue of $PATCH starts with a standard prologue, which specifies the file to be patched, whether a ZAP is to be produced, and the contents of the ZAP. The remaining dialogue depends on the mode in which you wish to patch the file: you can patch files by absolute addresses, offsets within fixed length records, addresses within compilation modules, or addresses within programs. As well as patching Global files, you can also patch non-Global volumes by specifying sector numbers.

2.7.1 Initial Dialogue

You are first prompted for details of the file to be patched and the print file to contain the listing of the ZAP. For example:

```
$98 FILE:P.SA UNIT:204
$98 ZAP FILE:<CR> UNIT:205
```

The default for the ZAP file is file-id Z.xxxxxx on unit $PR where xxxxxx is the first six characters of the filename. If you do not want to produce the ZAP corresponding to the patch (as will usually be the case during testing) you should reply <CTRL A> to the ZAP file prompt.

If you reply <CTRL B> to the file prompt this allows you to patch physical sectors of the volume on the specified unit, without reference to the Global directory structure. In particular, you can patch non-Global volumes.

If you specify a ZAP file then a number of further prompts appear. You are first prompted for up to 10 lines of title, to be printed at the start of the ZAP. The prompt is in the form of a text edit window under which the following line appears:

```
KEY UP TO 10 LINES OF ZAP TITLE
```

The use of keys in text editing is described in the Global Operating Manual in the section on $TED - the text edit utility. The edit is terminated by keying <ESCAPE> (which may need to be keyed twice on some screens).

Next you will be asked to specify whether the ZAP is to include the file-id of the file being patched (making it specific to one file), and whether to check the previous contents before patching the file (to help ensure that the correct version of the file is being patched):

```
$98 IS FILE-ID TO BE INCLUDED IN ZAP?:N
$98 IS ZAP TO CHECK PREVIOUS CONTENTS?:Y
```

We recommend you to make ZAPs check the previous contents, but not to include the file-id. In particular, do **not** include the file-id if you are patching a program or compilation library, as this will cause the ZAP to patch a fixed offset within the library, whereas if you omit the file-id from the ZAP it will patch the appropriate member name within the library, or indeed can be used to patch a stand-alone module.

2.7.2 The Patch Mode Prompt

You are next prompted for the patch mode you wish to use:

```
$98 PATCH MODE:
```

Your reply should be one of the following:

<ESCAPE>	to return to the file prompt;
<CTRL A>	to return to the file prompt;
A	to patch absolute file addresses;
C	to patch a compilation module or library;
P	to patch a program or program library;
R	to patch offsets within records (or offsets within sectors if patching the whole volume).

The dialogue for each patch mode is given in separate sections below. If the mode is inappropriate for the file specified, an error message will be displayed and the patch mode prompt repeated.

2.7.3 Patching Using Absolute Addresses

Using absolute patch mode (key A), the address to be patched is specified as a hexadecimal number of up to 8 digits, representing the byte number within the file counting from zero. When you specify a valid address, the corresponding byte in the file is displayed, in

hexadecimal with ASCII interpretation, and you are prompted to supply a new value:

```
$98 HEXADECIMAL FILE ADDRESS (00000000):124
00000124 41 'A':
```

You should make one of the following replies:

- a one or two digit hexadecimal value to change the byte to that value, and then display the next byte;
- a single quote (or <CTRL B>) followed by an ASCII character to change the byte to that character value, and then display the next byte;
- <CR> to leave the byte unchanged and display the next byte;
- <CTRL C> to leave the byte unchanged and display the previous byte;
- <CTRL A> or <ESCAPE> to return to the file address prompt.

A reply of <CTRL A> or <ESCAPE> to the file address prompt will return you to the file prompt. Note that you must return to the file prompt to ensure that the patch has been written to the file.

As an example, the dialogue:

```
$98 HEXADECIMAL FILE ADDRESS (00000000):124
00000124 41 'A':'B
00000125 20 ' ':<CR>
00000126 03 :2
00000127 B3 :<CTRL A>
$98 HEXADECIMAL FILE ADDRESS (00000124):<CTRL A>
$98 FILE:
```

will change byte #124 from ASCII A to B, and byte #126 from #03 to #02.

2.7.4 Patching Compilation Modules

If the file to be patched is a compilation library, you will first be prompted for the program name of the module to be patched:

```
$98 PROGRAM NAME:
```

Compilation patch mode allows you to patch any initialised byte within the compilation. The byte to be patched is specified as a location relative to the start of the compilation, as given on the compilation listing. If the specified location is part of a relocatable word, then the whole word is displayed as an offset from a global symbol, and you can either change the offset, or make the relocation relative to some other global defined within the compilation. You cannot however make relocatable items absolute or vice versa. If the address you specify is outside the compilation, or corresponds to an uninitialised byte, then the message NOT FOUND will be displayed and the location prompt is repeated.

Once you have specified the location, the byte or word at that location is displayed:

```
$98 HEXADECIMAL LOCATION (0000):1F
001F 24 '$' :
```

You should supply one of the following replies:

- a one or two digit hexadecimal value to modify the byte, and then display the next byte;
- a single quote (or <CTRL B>) followed by an ASCII character to change the byte to that character value, and then display the next;
- <CR> to display the next byte (or relocatable word), leaving the byte unchanged;
- <CTRL C> to display the previous byte or word, leaving the current byte unchanged;
- <CTRL A> or <ESCAPE> to return to the location prompt.

If the address specified is part of a relocatable word, the word is displayed in the form:

```
001C 0144[AR$A]:
```

meaning it is offset #144 from global AR$A. You should make one of the following replies to this prompt:

- a 1 to 4 digit hexadecimal offset to alter the offset;
- <CR> to leave the offset unchanged;
- <CTRL C> to display the previous byte or word;
- <CTRL A> or <ESCAPE> to return to the location prompt.

If you supply a value or key <CR> a second prompt will appear on the same line. You should reply either:

- with the name of a global defined in the compilation to change the global used in relocating the word; or
- <CR> to display the next byte (or relocatable word) and leave the global unchanged.

A reply of <CTRL A> or <ESCAPE> to the location prompt returns you to the file prompt. Note that you must return to the file prompt to ensure that all your patches have been written to the file.

If you key <CTRL C> to the location prompt $PATCH displays a further prompt:

```
$98 HEXADECIMAL OFFSET:
```

which allows you to patch any offset within the compilation module, including header, trailer and relocation records. The data in the compilation header is held in internal data format as a PIC 9(6) COMP field starting at offset #0A.

As an example, the dialogue:

```
$98 HEXADECIMAL LOCATION (0000):1C
```

```
001C 0144[AR$A ]:150 :<CR>
001E 06 ' ' :<CR>
001F 24 '$' :'*
0020 0A :<CTRL A>
$98 HEXADECIMAL LOCATION (001C):<CTRL A>
$98 FILE:
```

patches the relocatable word at location #1C to be offset #150 rather than #144 from global AR$A, and changes the character at location #1F from $ to *.

2.7.5 Patching Programs

When you use program patch mode, if the specified file is a program library you are first prompted for the program-id of the program to be patched:

```
$98 PROGRAM-ID:
```

Program patch mode allows you to patch any initialised bytes within the loadable program. You specify the location as the base address of the particular compilation to be patched (taken from the link map) and a location within the module (taken from the compilation listing). For example:

```
$98 HEXADECIMAL BASE ADDRESS:500
$98 HEXADECIMAL LOCATION (0000):21D8
26D8 [0500+21D8] 2A '*':
```

You should supply one of the following replies:

- one or two hexadecimal digits, to modify the byte to this value and then display the next;
- a single quote (or <CTRL B>) followed by an ASCII character, to modify the byte to the value of the character and then display the next;
- <CR> to leave the byte unchanged and display the next;
- <CTRL C> to leave the byte unchanged and display the previous byte;
- <CTRL A> or <ESCAPE> to return to the location prompt.

You can key <CTRL A> or <ESCAPE> to the location prompt to return to the base prompt, and <CTRL A> or <ESCAPE> to the base prompt returns you to the file prompt. Note that you must return to the file prompt to ensure that the patch has been written to the file.

If you key <CTRL C> to the base prompt $PATCH displays a further prompt:

```
$98 HEXADECIMAL OFFSET:
```

which allows you to patch any offset within the program file (or program module if inside a library). In particular, it allows you to patch the title, header and trailer records which indicate where the program is to be loaded.

If the program you are patching is relocatable, then you can still patch bytes, but you should note that any relocatable words within the program will be displayed as 2 bytes containing an offset, relative to

the start of the module, with the most significant byte first. You cannot patch relocatable words to be non-relocatable, or vice versa.

Note that if the location specified is not within the program, or corresponds to uninitialised data, then a message NOT FOUND will be displayed, and the location prompt repeated.

2.7.6 Patching Using Record Numbers

In record number patch mode, the file address is specified as a hexadecimal offset within a fixed-length record, whose number is specified in decimal. If the file is RS or IS, the actual record length is used. For other types of file, record number patch mode is still valid, a 'record length' of 256 bytes being assumed. This is the same length as is assumed by $L when inspecting a file, and hence record patch mode is the most convenient if you want to modify a part of a file which has been identified using $L.

You are first prompted for a record number (hexadecimal, counting from 1) and offset (decimal, counting from zero), and the byte thus identified is displayed in hexadecimal with an ASCII interpretation. For example:

```
$98 RECORD NUMBER ( 1):3
$98 HEXADECIMAL OFFSET IN RECORD (00000000):1A
3,0000001A 58 'X':
```

You should supply one of the following replies:

- one or two hexadecimal digits, to modify the byte to this value and then display the next byte;
- a single quote (or <CTRL B>) followed by an ASCII character, to change the byte to the value of the character and then display the next byte;
- <CR> to display the next byte, leaving the current byte unchanged;
- <CTRL C> to display the previous byte, leaving the current byte unchanged;
- <CTRL A> or <ESCAPE> to return to the record number prompt.

Note that the offset you specify is not restricted to being less than the record number: if it is greater the record number will be incremented accordingly, so that, for example, if you specify offset #100 in record 1, and the records are #100 bytes long, this gives you offset zero in record 2. This facility is particularly useful if you are examining records in the overflow area of an IS file.

A reply of <CTRL A> or <ESCAPE> to the record number prompt will return you to the file prompt. Note that you must return to the file prompt to ensure that the patch has been written to the file.

2.7.7 Patching Using Sector Numbers

If you keyed <CTRL B> to the file prompt, to patch physical sectors within the volume, then instead of the record number prompt a sector number prompt appears, followed by the offset prompt. For example:

```
$98 SECTOR NUMBER ( 1):27
```

```
$98 HEXADECIMAL OFFSET IN RECORD (00000000):
```

In effect, the volume is treated as an RS file whose record length is the sector size. If sectors within a track are assumed to be numbered from 1, and heads and tracks are numbered from zero, then the sector number as used by $PATCH is calculated as:

S + (H * SPT) + (T * SPT * HPC)

where:

S	Sector number
H	Head number
T	Track number
SPT	Sectors per track
HPC	Heads per cylinder (i.e. number of heads)

2.7.8 Notes on Patching Programs and Compilations

To understand fully the intermediate code generated by $COBOL you need to read Chapter 5, preferably in conjunction with a compilation listing produced with the binary list (BL) option. However, there are a number of simple patches that you can make during testing without needing to understand intermediate code completely.

If you want to patch out a Global Cobol statement, so that it has no effect, you should patch the first byte to #04, and the next byte to [length of the statement minus 2]. The length of the statement can be determined by subtracting its start location from that of the start of the next. Alternatively, for program files only, you can patch every byte of the statement to zero. (Note that these patches change the statement to be a $SET statement, which will have no effect unless you are using intermediate code in a non-standard way.)

If there is a literal as the first operand of an ADD, DO, IF, MOVE or MULTIPLY statement, or as the second operand of a DIVIDE or SUBTRACT statement, then this can be modified as follows. If the literal is a single character, or an integer in the range -128 to +127, it will be the second byte of the statement. Otherwise the literal starts at the third byte of the statement. Note that you cannot easily change the length of the literal.

You can patch a trap into a program by adding 1 to the value of the first byte of any statement. This can be useful in debugging complex overlay structures, particularly if they are LIVE linked. A trap set in this way can be cleared using the C instruction in $DEBUG.

If you want to modify the conditional test for a DO, IF or GOTO statement you must first identify the start of the transfer of control instruction, which is a byte set to #60 four bytes from the end of the instruction. The byte following (the qualifier) specifies the conditions under which the jump will occur, as follows:

0 = never	7 = always
1 = if zero	6 = if not zero
2 = if positive	5 = if not positive
4 = if negative	3 = if not negative
8 = if exception/overflow	16 = if no exception/overflow

and can be modified as required. Note that the two bytes following the qualifier are the address to which to jump (relocatable in the case of a compilation file).

2.8 $PDUMP - Formatted Program Dump Utility

The $PDUMP command produces a hexadecimal dump of a program file, with ASCII interpretation, formatted to show where in memory each text block will be loaded. Its main uses are to examine programs whose link map indicates that they are in more than one text block (and so cannot be quick loaded), in order to find out where the gaps are, and to check a program or program library for large areas which are initialised to zeros or spaces, and which could be changed to uninitialised data to reduce the size on disk.

To run the command you simply specify the filename and unit of the program to be dumped. For example:

```
GSM READY:$PDUMP
$217 INPUT FILE:SA100 UNIT:204
$217 LISTING UNIT:202
$217 INTERPRETING
GSM READY:
```

If you specify a program library, the whole of the library will be dumped. If you reply <CTRL A> to the listing unit prompt then the program will print to screen. Replying <CR> will sent the print to $PR.

2.9 $RECOVER - File Recovery Program

The file recovery program will attempt to recover relative sequential, indexed-sequential or text files that have become corrupt due to I/O errors, or by being only partially restored by $TDUMP, or by being overwritten with corrupt data due to a program or hardware error. In general, some data will be lost, and so you should check the recovered file very carefully; there might be other corrupt records which have not been noticed.

The recovery program can also be used to try to recover diskettes with I/O errors on the directory track, and to 'undelete' files which have been accidentally deleted.

When you run $RECOVER it first displays a menu of the available options:

You should select the appropriate option.

2.9.1 Accidentally Deleted Files

If you accidentally delete a valuable file, for example while using $F, it may be possible to 'undelete' the file using $RECOVER. You **cannot**, however, recover files from a volume which has been scratched or initialised. It will usually be possible to recover deleted files providing that you have not subsequently opened any new files on the volume. In particular, note that listing the directory results in a new file being opened, and so you should not list the directory before using $RECOVER. (It may, however, be possible to recover the file even if the directory has been listed.)

You will next be prompted for the unit containing the file to be recovered. You will then be presented with details of all the files it might be possible to recover, so that you can select the ones you want. When you recover a file, a prompt for a new file-id allows you to change its name, or a reply of <CR> recovers it with its original name. Note that there may be several copies of the deleted file in the directory, so you must select the correct one. If you are not sure, you can recover more than one copy, and inspect them later to determine the correct copy. For example:

```
$306 UNIT:204
$306 RECOVER B.SA1      TYPE TF     CREATED 10/11/84
       START 21348      SIZE 3421   EXTENT 3488?:N
$306 RECOVER S.SA1      TYPE TF     CREATED 11/11/84
       START 0          SIZE 3461   EXTENT 3488?:Y    NEW FILE-ID:<CR>
$306 RECOVER S.SA1      TYPE TF     CREATED 11/11/84
       START 3488       SIZE 3466   EXTENT 3488?:Y    NEW FILE-ID:S.2
$306 RECOVERY COMPLETE
GSM READY:
```

You must now use $INSPECT to examine the two files. It is quite likely that the incorrect version will not contain valid ASCII text, and will appear to be empty when inspected. Also, if you list the directory you may get the message 'INVALID DIRECTORY' at the end. This should disappear if you delete the incorrect version of the file.

2.9.2 Recovering a Diskette with a Corrupt Directory

If a diskette develops an I/O error within the directory, this option will allow you to recover all files except those whose directory entry was within the corrupted sector (typically 2 to 5). You must first initialise a new diskette (of the same format) onto which the files will be recovered.

You will be prompted for the units containing the corrupt disk and the new disk, then recovery will take place. For example:

```
$306 CORRUPT VOLUME ON UNIT:114
$306 NEW VOLUME ON UNIT:115
$306 RECOVERING
$306 FILES RECOVERED - UP TO 6 LOST
GSM READY:
```

Important Note: The diskette cannot be recovered if the first sector of the directory is unreadable.

2.9.3 Recovering Corrupt IS or RS Files

The corrupt IS or RS file recovery option allows you to recover most of the data in an IS or RS file which has become corrupt. The program can deal with three types of corruption: I/O errors; files partially restored by $TDUMP; and corrupt data. Normally a file would only have suffered one type of corruption, but it is possible to recover files with corruption of a mixture of types.

The recovery proceeds in two stages. Firstly, the areas of the file which are corrupt are identified. For corruption due to I/O errors or partial restoration by $TDUMP this is straightforward; for data corruption you must first identify the corrupted records, for example using the $L command, and supply the numbers of the corrupt records. Secondly, a new file is created on the output unit containing only valid records.

You are first prompted for the name and unit of the file to be recovered, and the name and unit for the recovered file. For example:

```
$306 CORRUPT FILE:SAMAST UNIT:114
$306 RECOVERED FILE:SAMAST UNIT:115
```

You will then be prompted for whether the file contains I/O errors or has been partially restored by $TDUMP:

```
$306 DOES FILE CONTAIN I/O ERRORS?:N
$306 HAS FILE BEEN PARTIALLY RESTORED BY $TDUMP?:
```

In either case, if you key Y the file will be scanned for corrupt blocks, and messages of the form:

```
$306 PRIME DATA RECORDS 124 TO 127 CORRUPT
```

or:

```
$306 OVERFLOW RECORDS 11 TO 12 CORRUPT
```

or:

```
$306 RECORD NUMBERS 237 TO 238 CORRUPT
```

will be displayed. It may be possible, by using $L, to identify from this information which records have been lost. It may be that the corruption is outside the allocated file space, or is entirely within the index of an IS file. In this case it is possible to completely

recover the file by using the $CONV command to create a new copy of the file, and the message:

```
$306 NO DATA RECORDS ARE CORRUPT
$306 FILE CAN BE RECOVERED USING $CONV
```

will be displayed.

Finally, you can specify records which contain data corruption. These must be identified in advance, for example by inspection using $L. For IS files, you must first specify any corrupt prime data records, and then any corrupt overflow records (numbering both from 1). For example:

```
$306 PRIME DATA RECORDS CORRUPT FROM:23 TO:24
$306 PRIME DATA RECORDS CORRUPT FROM:<CR>
$306 OVERFLOW RECORDS CORRUPT FROM:<CR>
```

In both cases, the prompt is repeated until you key <CR>, and the ranges are inclusive. For RS files, you need only specify the numbers of the corrupt records. For example:

```
$306 RECORDS CORRUPT FROM:23 TO:24
$306 RECORDS CORRUPT FROM:<CR>
```

Recovery of the file will then take place:

```
$306 RECOVERING
$306 RECOVERY COMPLETE
GSM READY:
```

2.9.4 Text File Recovery

Recovery of text files consists simply of replacing non-ASCII characters by ? characters and replacing any blocks containing I/O errors by lines consisting of 30 ? characters. This will produce a file which can then be corrected by use of the editor, $EDIT.

You are first prompted for the name and unit of the corrupt file, and of the new file to be created:

```
$306 CORRUPT FILE:SAPROG UNIT:114
$306 RECOVERED FILE:SAPROG UNIT:115
```

A prefix of S. is assumed. File recovery will then take place:

```
$306 RECOVERING
$306 RECOVERY COMPLETE
GSM READY:
```

2.10 $SCOMP - Source File Comparison Utility

The source comparison utility takes two versions of a source file and prints out lines that differ. It identifies blocks of text that have been inserted or deleted by searching through the sources for lines further on that match. Any discrepancies are printed on a listing, the left hand side of which is text from the first input file, and the right hand side the corresponding text from the second file. Thus, for example, if the second file contains a new block of text which has been inserted, this will be printed on the right of the listing, with nothing on the left. Lines which are the same in the two files are not printed.

2.10.1 The Source Type Prompt

When you run $SCOMP, it first prompts you for the type of source file to be compared, for example:

```
GSM READY:$SCOMP
$303 SOURCE TYPE (PROG):
```

You can reply <CTRL C> to list out the different types of source files supported. Reply with a source type, or <CR> to accept the default of comparing Global Cobol program sources. This information about the source type is used to improve the efficiency of the matching algorithm, used when a discrepancy is found, by avoiding spurious matches on common lines such as an EXIT statement in a Global Cobol program.

The utility is mainly used to determine what updates have been made to a source since some earlier version. For example, it can be used to list out the changes made to convert a UK version of a product to an overseas version.

2.10.2 The Option Prompt

Once you have supplied the source file type, an option prompt is displayed:

```
$303 OPTION:
```

You can reply <CTRL C> to list the available options. The initial options are to compare all the lines of the source. However, you can choose to select only parts of the source files for comparison.

2.10.3 The File Prompts

When you have specified any options required, reply <CR> to the option prompt and the file prompts will appear. For example:

```
$303 OLD FILE:SAPR1 UNIT:203
$303 NEW FILE:SAPR2 UNIT:204
```

A prefix of S. is assumed. If you reply <CR> to either of the new file prompts, the old file name or unit will be assumed. If you reply <CTRL B> to the old file name prompt then you can compare all the S. files on two units against each other.

Further prompts of the form:

```
$303 COMPARE UNIT:203 WITH UNIT:204
```

will appear.

2.10.4 The Listing Unit Prompt

Once the files have been specified, a listing unit prompt appears:

```
$303 LISTING UNIT:
```

Reply with the unit to which the listing is to be directed, <CR> to print the listing on unit $PR, or <CTRL A> to display the differences on the screen.

A message of the form:

```
$303 COMPARING FILES name [name]
```

then appears, and when the comparison is complete the message:

```
$303 COMPARISON COMPLETE - IDENTICAL
```

or:

```
$303 COMPARISON COMPLETE - DIFFERENT
```

is displayed. Once all the files have been compared, the old file prompt is redisplayed. You supply the name of a further file to be compared, reply <CTRL A> to return to the source type prompt, or key <ESCAPE> to quit.

2.11 $STACK - Stack Maintenance

The $STACK command enables you to change the sizes of the user and system stacks and to load, unload and allocate relocatable programs and data items on them.

The RESOLVING and DISSOLVING prompts are for use by developers at TIS Software Ltd and <CR> should always be keyed to them.

The twelve stack maintenance instructions and three miscellaneous instructions available in $STACK are displayed when you key <CTRL C>, as shown below:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:<CTRL C>
LOU LOT LOS UNU UNT
UNS ENU ENT ENS DAU
DAT DAS LIS PRI SIZ
$219 STACK MAINTENANCE
```

The use of these instructions is explained below:

2.11.1 LOU - Load User Stack

This instruction is used to load a relocatable program on the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:LOU :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.2 LOT - Load Temporary User Stack

This instruction is used to load a relocatable program on the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:LOT :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.3 LOS - Load System Stack

This instruction is used to load a relocatable program on the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:LOS :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.4 UNU - Unload User Stack

This instruction is used to unload a module from the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:UNU :$MODULE DISSOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.5 UNT - Unload Temporary User Stack

This instruction is used to unload a module from the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:UNT :$MODULE DISSOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.6 UNS - Unload System Stack

This instruction is used to unload a module from the indicated stack. The dialogue is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:UNS :$MODULE DISSOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.7 ENU - Determine entry point of a module on the User Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:ENU :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.8 ENT - Determine entry point of a module on the Temporary User Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:ENT :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.9 ENS - Determine entry point of a module on the System Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:ENS :$MODULE RESOLVING:<CR>
$219 STACK MAINTENANCE
```

2.11.10 DAU - Allocate a data item on the User Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:DAU :$MODULE SIZE:size of item
$219 STACK MAINTENANCE
```

2.11.11 DAT - Allocate a data item on the Temporary User Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:DAT :$MODULE SIZE:size of item
$219 STACK MAINTENANCE
```

2.11.12 DAS - Allocate a data item on the System Stack

The dialogue for this instruction is as follows:

```
GSM READY:$STACK
$219 STACK MAINTENANCE
:DAS :$MODULE SIZE:size of item
$219 STACK MAINTENANCE
```

2.11.13 LIS - List details of User and System Stacks

This command lists on the screen the size and contents of both stacks.

2.11.14 PRI - Print details of User and System Stacks

This command prints on unit $PR the size and contents of both stacks.

2.11.15 SIZ - Change size of User Stack

This instruction allows you to change the size of the user stack by specifying the new size or the amount you want to reduce it by. The dialogue is as follows:

```
$219 STACK MAINTENANCE
:SIZ
User area size currently 39548 bytes
Key new size or reduction, <CR> to exit:25000
User area size currently 25000 bytes
Key new size or reduction, <CR> to exit:100
User area size currently 24900 bytes
Key new size or reduction, <CR> to exit:<CR>
$219 STACK MAINTENANCE
:<ESC>
:GSM READY:
```

2.12 $URMAIN - User System Request Data Library Maintenance

The $URMAIN utility is used to create and update records within the end user system request reference data library, $$UREQ. For full information on how to create user system requests see the Global Development System Subroutine Manual. The $$UREQ data library consists of records each containing information about a particular end user system request.

2.12.1 The Library Prompt

When you run $URMAIN it prompts you for the end user request library an unit:

```
Key library name:$$UREQ Unit:201
```

2.12.2 The Option Prompt

After you have supplied the library name and unit the following prompt will appear:

```
Key Create, Amend, Print, Title, <ESC> to exit:
```

2.12.3 Creating a new record

To create a new record you key C to the option prompt. You are now asked for the new record name which you must supply. If a record of that name already exists you will be asked if you want to delete it. You will now enter the record detail screen and are prompted for the record information; the record title; the system request program and unit; the system request library (if any); the mode and the data set.

The record will be written to the system request library. If the library is full then you will be given an option to delete an existing record in the library or alternatively to abandon this record by keying E to end.

2.12.4 Amending an existing Record

To amend an existing record you must key "A" to the options prompt after which you will be prompted for the record name. You can optionally key "?" to the record name prompt which will allow you to list the current library. You can then select a record to amend.

You will now enter the record details screen as for the create option, and you can amend the record information.

2.12.5 Printing the Library

To print a listing of the contents of the data library you must key P to the options prompt. You will now be prompted for the listing unit on to which the report will be printed.

2.12.6 Amending the Library Title

To change the system request library title you must key T to the options prompt after which you will be asked for the new title as follows:

```
Title:Development System Requests
```

2.13 $VOL - List Volume Directory Utility

The $VOL utility produces an expanded directory listing of a volume, combining a listing of the files, as given by $F, with details of the contents of libraries. Thus the listing produced by $VOL gives a full listing of all the programs and compilations a volume contains, whether they appear as separate files or within a library.

Copy libraries are also analysed to produce a listing of the contents. Certain assumptions are made about the format of the copy library: there are assumed to be no lines other than comments and PAGE statements preceding the first book; if there are lines preceding the first book the first such line is assumed to contain the library title; any text following the book name is taken as the title of the book.

2.13.1 The Output Unit Prompt

When you run $VOL, it first prompts you for an output unit:

```
$505 LISTING UNIT:
```

Reply <CR> if you want the listing printed on $PR, <CTRL A> if you want it displayed on the screen, or enter a unit-id to cause the listing to be printed on that unit.

2.13.2 The Option Prompt

After you have supplied the output unit, an option prompt will appear:

```
$505 DIRECTORY OPTION:
```

Reply <CTRL C> to list the available options. Normally, the default options are sufficient, and you should key <CR> to use these.

2.13.3 The Title Prompt

When you reply <CR> to the option prompt, a title prompt appears:

```
$505 TITLE (UP TO 30 CHARS):
```

If you supply a title, of up to 30 characters, this will be printed on each page of the listing. If you reply <CR>, the title will be left blank.

2.13.4 The Unit Prompt

When you have supplied a title, a unit prompt appears. You should mount the volume to be listed, and supply its unit address. When complete, a message will be displayed and the prompt repeated so that you can list further volumes.

If you reply <CTRL A> to the unit prompt, the listing unit prompt will be redisplayed.

Reply <CR> or <ESCAPE> to the unit prompt in order to quit.

2.14 $XBUILD and $XPRINT - Global Cross-Reference Generators

The two Global Cross-Reference commands, $XBUILD and $XPRINT, are used to merge together a number of cross-reference listings, produced by $XREF, into a single global cross-reference. You can also merge details of references made within maps produced by Screen Formatting. By using these commands to merge cross-references and map files for all programs in a system, you can, for example, quickly determine which programs reference a particular field.

It is also possible to merge together a number of link maps and produce a cross-reference of these. This listing will, for example, allow you to determine which programs will need relinking if you change a root module in an overlay structure. Note that these references may for convenience be added to the file used for the global cross-reference, but are logically completely separate.

To produce a global cross-reference, you must go through three stages:

- run the $XREF utility on each program, writing the listing to disk or diskette;

- run the $XBUILD command to merge the cross-reference listings, maps and link maps into a single file;

- run the $XPRINT command to select the items to be cross-referenced, sort them into the correct sequence, and print them.

You can, if you wish, run $XPRINT again with different options to produce a cross-reference of fields in copy books, sorted starting with the 3rd character of each name. This is useful if the first two

characters of such names have been replaced using a COPY SUBSTITUTING statement, since it groups all references to the field together.

The file produced by $XBUILD can be very large, so you will need to make a large empty disk volume available. Furthermore, a sort work file of equal size will also be needed. The production of a global cross-reference can take several hours, particularly if the listings are on diskette. As a guide, to produce a global cross-reference of the whole of the Global System Manager subroutines and command programs involves approximately 800 separate programs, produces a file of about 7 Mbytes, and takes about 7 hours once the individual cross-reference listings have been produced.

2.14.1 Creating the Cross Reference Listings using $XREF

You must first produce cross-reference listings of each individual program by using the $XREF command, and write the listing file to disk or diskette. These cross-references must be produced using the no long name (NLN) option. You **cannot** use the LN (30 character names) option with $XBUILD.

If the SN (cross-reference by section name) option is specified these will appear in the global cross-reference listings. If the NSN (no section names) option is specified, the section names will appear as spaces.

2.14.2 Building the Merged Cross-Reference file using $XBUILD

Once all the individual cross-reference listings (and any link maps) have been produced they can be merged into a single file using the $XBUILD command. This file will contain a 19 byte record for each definition and reference; make sure that enough space is available on the output unit. If you allow the same amount of space as the cross-reference listings occupy, this should be more than sufficient.

When you run $XBUILD it first prompts you for the output unit:

```
$301 PLEASE KEY OUTPUT UNIT:
```

If a merged cross-reference file 'XREF' does not already exist on the output unit then you are asked:

```
$301 NEW XREF FILE?:
```

Type Y to create XREF.

You will next be prompted for an input unit, and whether files on this unit are to be deleted after they have been processed, for example:

```
$301 PLEASE KEY INPUT UNIT:100
$301 ARE INPUT FILES TO BE DELETED AFTER PROCESSING?:
```

If you reply <CTRL A> to the input unit prompt then you are asked:

```
DELETE RECORDS FOR FILE:
```

You can now enter a list of up to twenty files whose data is to be deleted. This is useful if you have included a new XREF and need to replace it with an updated version and is much quicker than doing each file individually. Key <CR> to the input unit prompt to continue. You will then be asked for the address of the work unit.

If you reply N to ARE INPUT FILES TO BE DELETED AFTER PROCESSING then each cross-reference listing, screen formatting map (possibly in a library) or link map listing on the input volume will be merged into the output file without deleting the input files, and when they have all been processed a prompt:

```
$301 MOUNT NEXT INPUT VOLUME ON unit:
```

will appear. Either mount the next volume on the input unit and reply <CR> or Y, or if there are no more input volumes to be processed at present key N to exit. If for any reason the program fails the output file will contain records for those input volumes which have been completely processed, but will not contain any records for the volume being processed.

If you reply Y to the 'ARE FILES TO BE DELETED' prompt this indicates that the input unit specified is a work unit, to which the files to be cross-referenced will be copied. As each file on the unit is processed, it is deleted, so that rather than having to change the input volume at regular intervals, you can copy further listings onto the work unit as convenient (using another partition of multi-user system), and keep the program running continuously. When all the files on the input unit have been processed, the prompt:

```
$301 IDLE - QUIT?:
```

appears. Either copy further files onto the work unit and reply <CR> or N, or if there are no further files to be processed reply Y. If the program fails the output file will contain records only for those input files which have been fully processed and deleted - hence if you restart $XBUILD after a failure it will continue processing correctly.

2.14.3 Printing a Global Cross Reference using $XPRINT

When you have produced the merged cross-reference file using $XBUILD, you can then run $XPRINT to print out the global cross-reference listing. It first prompts you for the input unit and a sort work unit, for example:

```
$302 PLEASE INPUT UNIT:204
$302 SORT WORK UNIT:205
```

The listing will be written to unit $PR. The input file will be truncated to release any free space. The sort work unit must have space for a work file at least as large as the merged cross-reference file produced by $XBUILD. The default sort work unit, if you key <CR>, is the input unit.

A menu of options will then be displayed. When you select one, the sort and print will start.

The 'every reference' option gives all references except link map references, sorted on program name within field name. The 'copied items sorted on third character' option cross-references only items in copy books, and sorts them on characters 3 to 6 of the field name, then on characters 1 and 2, then on program name. Thus if you have used 'COPY SUBSTITUTING' to change the first 2 characters of a copy book, references to these fields will be grouped together in the listing.

The 'link map items' option selects just those entries which come from link maps, and sorts them by program name within global name. The listing indicates whether references are in the program proper or in an information overlay.

3. Product Translator

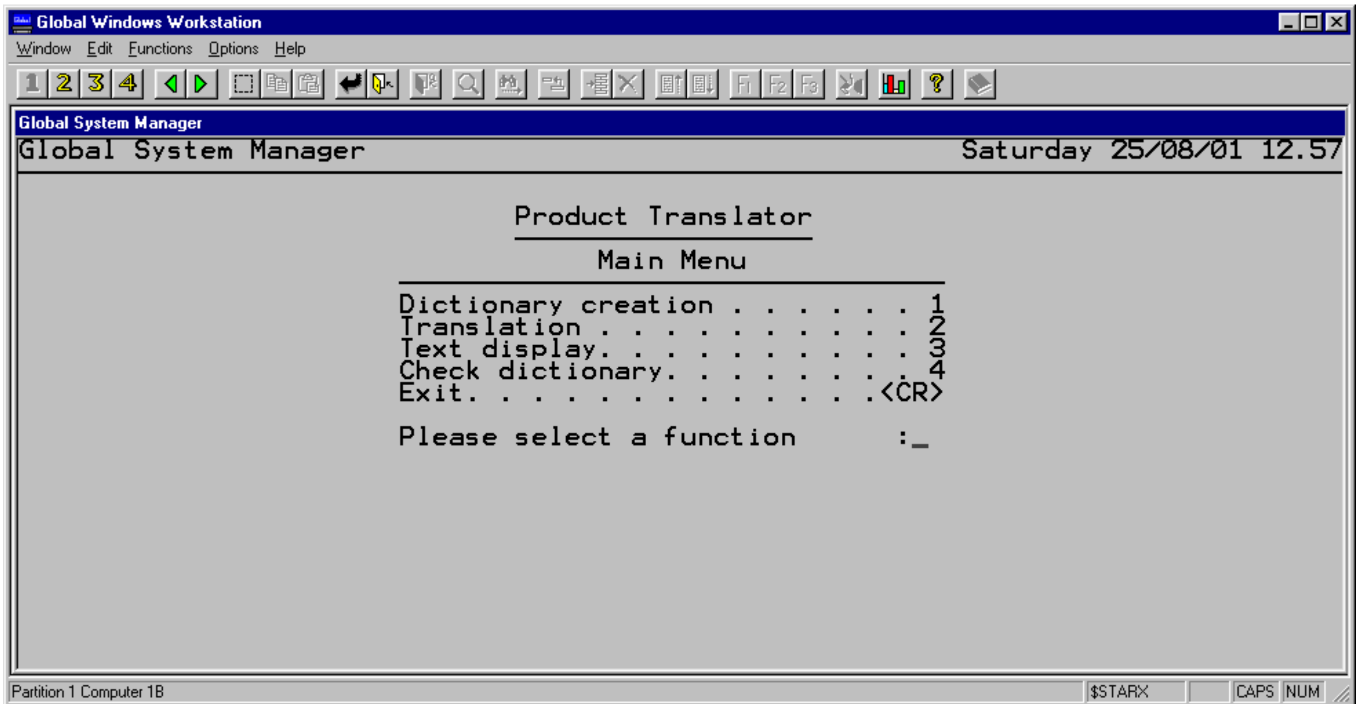


Figure 3.1 - The Product Translator Menu

3.1 Introduction

3.1.1 Overview

The Product Translator provides the facility to alter the text of messages and prompts in Global Cobol programs directly, without having to use the normal Edit, Compile and Link procedures. This allows you to change the text easily whilst leaving the logic of the program untouched.

The text in the program is altered in two separate stages. Firstly, a dictionary is created by scanning the program for text. The dictionary is a standard Global text file and can be processed by any of the appropriate Global System Manager and Global Cobol utilities. The dictionary contains each text string selected for translation together with the replacement string specified during the creation of the dictionary or subsequent editing. The second stage is the translation of the text strings within the program itself, using the dictionary for the appropriate translations.

The Product Translator Menu provides four options, these being:

Creating a Dictionary This scans the program, allowing you to create a set of words or phrases (referred to as a 'dictionary') in a text file together with their corresponding translations;

Translation This scans the program again, replacing the specified dictionary translations;

Text Display This allows you to examine the text contained within a program, without creating a dictionary or altering the text;

Check Dictionary This option allows you to check the text strings contained within a dictionary against those contained within the program to ensure that they match.

3.1.2 Installation

The Product Translator is installed by copying the file PTRANS from the appropriate Toolkit distribution diskette to a program unit. To run the program, set up a menu entry for it or else key its program-id, PTRANS, in response to the GSM ready or main menu selection prompt and reassign $P as follows:

```
GSM READY:PTRANS<CTRL A>
PLEASE ASSIGN $P:PTRANS unit address
```

3.2 Creating a Dictionary

For the purposes of product translation, a dictionary is a text file that contains a set of words or phrases that appear within a program together with their corresponding translations. This dictionary must be created before the actual translating of a program's text can be done. To create a dictionary, select the Dictionary Creation option from the Product Translator menu.

3.2.1 Specifying the Library or Program

The first prompt asks you for the file-id of the library containing the program to be processed and its unit address. If you do not explicitly key a prefix, the prefix 'P.' will be assumed by default. For example:

```
LIBRARY:AC UNIT:204
```

indicates that library P.AC on unit 204 is to be processed.

If you reply <CR> to the first part of the library prompt then this means that the program to be processed is not held in a library. To return to the Product Translator menu key <ESCAPE>.

After you have replied to the library prompt, you are then asked for the program-id of the program to be processed. Key <CTRL A> to return to the library prompt. If you specified that the program was not held in a library then you will also be prompted for its unit address. For example:

```
LIBRARY:<CR>
PROGRAM:SAMPLE UNIT:204
```

Key <ESCAPE> to either prompt to return to the menu. If you did specify a library in response to the library prompt, and supplied a unit address for the library, then you will only be prompted for the program-id. In this case, you may key <CR> if you wish all members of the library to be processed in turn. For example:

```
LIBRARY:AC UNIT:204
PROGRAM:<CR>
```

requests that all the members of library P.AC on unit 204 should be processed.

3.2.2 Specifying the Dictionary

Next you will be prompted for the file-id of the dictionary which will hold the text strings and their replacements. If you do not explicitly key a prefix, then the prefix 'T.' will be assumed as a default for the file-id. For example:

```
DICTIONARY (T.AC):SAMPLE UNIT (204):<CR>
```

specifies that the dictionary is file T.SAMPLE on unit 204.

If you specified that all members of a library were to be processed, a <CR> reply to the dictionary prompt specifies a file-id constructed by replacing the 'P.' prefix of the library name by the 'T.' prefix. If you specified a single program to be processed, a <CR> reply to the dictionary prompt specifies a file-id of 'T.' followed by the program name.

You may key <CR> to the dictionary unit prompt instead of supplying a unit address if this is the same as the program unit.

3.2.3 The Option Prompt

The Option prompt is now displayed. This allows you to specify which, if any, of the available options you wish to use:

```
OPTION, ? FOR HELP:
```

You may key <CTRL A> to cancel all options previously specified and re-specify standard processing.

If you key ?, a list of options and their meanings will be displayed, with the message 'IN FORCE' indicating which options have been selected.

When <CR> is keyed to the option prompt, processing continues in accordance with the specified options (or standard processing takes place if no options have been specified).

The options available when you are creating a dictionary are as follows:

3.2.3.1 A - Automatic Dictionary Creation

When this option is in effect, the Product Translator outputs each text string selected during a search to the dictionary, followed by the text string or standard replacement without displaying the string or prompting you. You can check the progress of the creation by keying <CTRL G>. The name of the library will be displayed, if one is in use, followed by the name of the program being processed and the last string found by the search.

3.2.3.2 E - Edit Text During Creation

If you specify this option, the effect of keying an asterisk in a replacement string is modified, so that part of the string displayed is output to the dictionary but, instead of being followed by a partial replacement, it is followed by a blank line.

3.2.3.3 I - Ignore Short Strings

There is no way that the Product Translator can reliably differentiate between text to be translated and program instructions that happen to correspond to a sequence of ASCII letters. The I option allows you to specify the minimum length of text string to be processed, for example:

```
OPTION:I  MINIMUM NUMBER OF CHARACTERS TO BE PROCESSED:5
```

specifies that strings of less than 5 are to be ignored. The longer the minimum string, the smaller the chance of finding spurious 'words' in the program code. In general, it is best to accept only strings of 4 or more characters or else large numbers of short, spurious strings will appear in the dictionary.

You will now be prompted:

```
PROCESS SINGLE CHARACTER REPLIES?(N):
```

Certain single characters embedded within the procedure division code can be identified fairly reliably by the preceding instruction code; this option allows you to translate these (they correspond to single characters in MOVE or IF statements). Key Y if you wish to process single character replies or <CR> if you do not.

3.2.3.4 L - Lower Case Included

This option modifies the search so that, once an upper case letter has been found at the start of a new text string, lower case letters (ASCII codes from hex 61 to hex 7A) are included as continuation characters of the string, in addition to upper case letters and spaces. When you select this option, the prompt:

```
AND AS START OF STRING?(Y):
```

is output. You should key Y or <CR> to specify that a lower case character may also be taken as the start of a new string. Key N if you only wish lower case characters after the first character to be processed.

3.2.3.5 M - Monitor Format Expected

If the program to be searched or translated is the GSM monitor ($MONITOR) then you must specify this option to instruct the Product Translator to process the special program file format of the monitor.

3.2.3.6 N - Numbers Included

This option modifies the search so that numbers (ASCII codes from hex 30 to hex 39) may be included as continuation characters of a string. When you select this option, the prompt:

```
AND AS START OF STRING?(Y):
```

is output. You should key Y or <CR> to specify that a number may also be taken as the start of a new string. Key N if you only wish numbers after the first character to be processed.

3.2.3.7 R - Replace Standard String

This option allows you to specify a standard replacement string for a string which occurs frequently in the program. For example:

```
OPTION, ? FOR HELP      :R
INPUT STRING            :DAY
REPLACEMENT STRING      :TAG
```

3.2.3.8 S - Special Characters Included
This option modifies the search so that special characters (i.e. characters that are neither alphabetic nor numeric) may be included as continuation characters of a string. If you specify this option, the numbers included option is automatically invoked. When you select this option, the prompt:

```
KEY SPECIAL CHARACTERS:
```

followed by a field editable list of all the special characters is displayed. If you want all of these special characters to be included then key <CR>, otherwise key in the specific characters you require. A further prompt:

```
SPECIAL START CHARACTERS:
```

is then displayed followed by a field editable list of all the special characters. You should amend the list so that it contains those character you require.

3.2.3.9 T - Trailing Spaces Not Removed
When the text strings to be replaced have trailing spaces which are also to be replaced you should specify this option to prevent their removal. Trailing spaces in the original text string or the replacement string will be converted to hash characters when the strings are displayed or output to the dictionary.

3.2.3.10 W - Single Words
This option modifies the search so that spaces are not included as continuation characters of the string.

3.2.4 Creating the Dictionary
When you have keyed <CR> to the Option prompt, the Product Translator searches the program for text strings whose first character is an upper case letter (ASCII codes from hex 41 to hex 5A) and whose subsequent characters are either upper case letters or spaces. Each string is displayed between asterisks with trailing spaces removed. You will then be prompted for a replacement string. For example:

```
*SAMPLE TEXT*
:REPLACEMENT
```

specifies that the string 'SAMPLE TEXT' is to be replaced by the string 'REPLACEMENT'.

If you key a replacement string shorter than the string displayed, the replacement string will be padded with spaces. However, if you key a shorter replacement string terminated by an asterisk, this will be taken as a replacement string for part of the string displayed and the remainder will be redisplayed as follows:

```
*SAMPLE TEXT*
:PARTIAL*
*TEXT*
:
```

In each case, the displayed string, or a part of it, will be output to the dictionary followed by the replacement string on the next line.

3.3 Translation

Translation is the process whereby text strings in the subject library or program are replaced by the replacement strings specified for them in the dictionary created in the dictionary creation phase.

3.3.1 Specifying the Library or Program

The first prompt asks you for the file-id of the library containing the program to be processed and its unit address. If you do not explicitly key a prefix, the prefix 'P.' will be assumed by default. For example:

```
LIBRARY:AC UNIT:204
```

indicates that library P.AC on unit 204 is to be processed.

If you reply <CR> to the first part of the library prompt then this means that the program to be processed is not held in a library. To return to the Product Translator menu key <ESCAPE>.

After you have replied to the library prompt, you are then asked for the program-id of the program to be processed. Key <CTRL A> to return to the library prompt. If you specified that the program was not held in a library then you will also be prompted for its unit-address. For example:

```
LIBRARY:<CR>
PROGRAM:SAMPLE UNIT:204
```

Key <ESCAPE> to either the library or the program prompt to return to the menu. If you did specify a library in response to the library prompt, then you will only be prompted for the program-id. In this case, you may key <CR> if you wish all members of the library to be processed in turn. For example:

```
LIBRARY:AC UNIT:204
PROGRAM:<CR>
```

requests that all the members of library P.AC on unit 101 should be processed.

3.3.2 Specifying the Dictionary

Next you are prompted for the file-id and unit address of the dictionary you have established. If you do not explicitly key a prefix, then the prefix 'T.' will be assumed as a default for the file-id. For example:

```
DICTIONARY (T.AC):SAMPLE UNIT (204):<CR>
```

specifies that the dictionary is file T.SAMPLE on unit 204.

If you specified that an entire library was to be processed, a <CR> reply to the dictionary prompt specifies a file-id constructed by replacing the 'P.' prefix of the library name by the 'T.' prefix.

If you specified a single program to be processed, a <CR> reply to the dictionary prompt specifies a file-id of 'T.' followed by the program name.

You may key <CR> to the unit prompt instead of supplying a unit-id if this is the same as the program unit.

3.3.3 Translating the Program

The text in the library or program will now be translated. The progress of the translation can be checked by keying <CTRL G>; the name of the library will be displayed, if one is in use, followed by the name of the program and the last string found by the search.

The Product Translator proceeds by reading records from the dictionary. Records whose first characters are asterisks are treated as comments. When a record is read that is not a comment, the Product Translator will search the specified program for the text string contained in the record. If the text is found, the dictionary is read again and the text in the program is replaced by the replacement string from the next record that is not a comment. Trailing hash characters in the original text or the replacement string are treated as spaces. This sequence continues until the end of the dictionary is encountered. If a text string cannot be found in the program, an error message is displayed and the program is terminated.

When the program has been successfully translated the message 'TRANSLATED' will be displayed.

3.4 Text Display

It may often be useful to be able to examine the text in a program without going on to create a dictionary or translate the text. To do this select option 3 from the Product Translator menu.

3.4.1 Specifying the Library or Program

The first prompt asks you for the file-id of the library containing the program to be processed and its unit address. If you do not explicitly key a prefix, the prefix 'P.' will be assumed by default. For example:

```
LIBRARY:AC UNIT:204
```

indicates that library P.AC on unit 204 is to be processed.

If you reply <CR> to the first part of the library prompt then this means that the program to be processed is not held in a library. To return to the Product Translator menu key <ESCAPE>.

After you have replied to the library prompt, you are then asked for the program-id of the program to be processed. Key <CTRL A> to return to the library prompt. If you specified that the program was not held in a library then you will also be prompted for its unit address. For example:

```
LIBRARY:<CR>
PROGRAM:SAMPLE UNIT:204
```

Key <ESCAPE> to either prompt to return to the menu. If you did specify a library in response to the library prompt, then you will only be prompted for the program-id. In this case, you may key <CR> if you wish all members of the library to be processed in turn. For example:

```
LIBRARY:AC UNIT:204
PROGRAM:<CR>
```

requests that all the members of library P.AC on unit 204 should be processed.

3.4.2 The Option Prompt

The Option prompt is now displayed. This allows you to specify which, if any, of the available options you wish to use:

```
OPTION, ? FOR HELP:
```

You may key <CTRL A> to cancel all options previously specified and re-specify standard processing.

If you key ?, a list of options and their meanings will be displayed, with the message 'IN FORCE' indicating which options have been selected.

When <CR> is keyed to the option prompt, processing continues in accordance with the specified options (or standard processing takes place if no options have been specified).

For a text display the following options are available:

- I   Ignore Short Strings
- L   Lower Case Included
- M   Monitor Format Expected
- N   Numbers Included
- S   Special Characters Included
- T   Trailing Spaces Not Removed
- W   Single Words

These options are as explained on page 3-7.

3.5 Check Dictionary

The fourth option in the Product Translator menu allows you to check the text strings contained within a dictionary against those contained in a program. This can be useful if you have been editing the dictionary (e.g. amending translations) and wish to ensure that you have not accidentally deleted any of the text strings.

As with the other options you will be prompted for the necessary library, program and unit of the program and dictionary. Once these have been keyed in and accepted, the dictionary will be checked against the program and if it is correct the message:

```
DICTIONARY CHECK SUCCESSFUL
```

will be displayed.

3.6 Error Messages

NOT FOUND OR WRONG TYPE

Either the file whose file-id you specified in response to a file prompt was not present on the unit specified or it was present but was not of the appropriate file organisation.

FILE ALREADY EXISTS - DELETE?

A file with the same file-id as the dictionary you are about to create exists on the unit specified. You should key Y if you wish to delete the file. Any other reply will leave the existing file undisturbed.

INVALID - REINPUT

Re-key the option, after keying ? for a list of options if necessary.

TEXT NOT FOUND IN xxxxxxxx

The text string specified was present in the dictionary being used to translate the named program but the string was not found in the program. Check that the program and dictionary were correctly specified and that the dictionary has the correct format.

INVALID DICTIONARY

A dictionary record was encountered which has an incorrect newline sequence. Check that the dictionary name was correctly specified and, if so, recreate the dictionary.

INVALID DICTIONARY - UNEXPECTED END OF FILE

Unexpected end of file occurred when attempting to read a dictionary. Check that the dictionary name was correctly specified. If it was, check that each text string is followed by a replacement string.

REPLACEMENT STRING LARGER THAN ORIGINAL

A dictionary record containing a replacement string was larger than the preceding record containing the string to be replaced.

4. Intermediate Code Language

4.1 Foreword

Intermediate Code is a business application oriented language for micro and minicomputers and is the language into which Global Cobol programs are translated. The instruction set supports fixed point arithmetic; input/output conversion; character string handling; one dimensional arrays; pointer variables and based variables; and a subroutine calling mechanism with 'call by name' parameters. In addition the language has sophisticated exception handling which enables exceptions either to be processed by the application program or to be passed to a monitor program, itself written in intermediate code.

The language is like a powerful single address assembly language. For example, only one instruction is needed to convert a binary number to a printable, signed character string with a specified number of places before and after the decimal point. Yet on the other hand, an arithmetic assignment statement such as:

```
Z = X x Y
```

cannot be processed by a single instruction. Instead, the following sequence is required:

- Load accumulator with X;
- Multiply accumulator by Y;
- Store accumulator in Z.

Intermediate code instructions are executed by an interpreter. An assembler language routine (the steering routine) passes control to the interpreter, giving it the address of location 0 of the program address space. Pointers to various routines and control blocks must be set up before control is passed to the interpreter: in particular, a pointer to the start of the monitor program which is to be entered at the beginning of the interpretation and also whenever a program exception occurs. Control is only returned to the steering routine if a terminal error occurs (which should not happen in a debugged system) or if an exit is made from the monitor program.

Intermediate code routines are produced as relocatable code modules which can be linked together using the Global Cobol linker. Programs are always linked to execute in an address space starting at zero: they are dynamically relocated by the interpreter when they are executed. This means that programs do not have to be relinked to run in different address spaces.

Note that in the description which follows intermediate code instructions are always referred to as proper nouns, beginning with a capital letter. For example:

```
Move
Stop
Exit
```

Global Cobol statements are always shown in block capitals:

```
MOVE A TO B
STOP
EXIT
```

4.2 The Intermediate Code 'Computer'

Because it is interpreted, intermediate code is best understood if it is considered to be executing on a virtual computer. Therefore any document which describes intermediate code and its interpreter properly begins by describing the architecture of this computer. By 'architecture' is meant the data formats and instruction formats the computer recognises; the registers and data areas used by its CPU (the interpreter itself); and the way special conditions such as initiation and termination, program errors and the like, are handled.

This chapter is devoted to the computer architecture. The remaining chapters describe the individual instructions, each chapter being devoted to a particular class of instructions.

Size in bytes	**Capacity x 10^s (s is the number of digits following the decimal point)**	
	Exact	**Approximate**
1	-128 to 127	
2	-32768 to 32767	
3	-2^{23} to $2^{23}-1$	$\pm 8.389 \times 10^{6}$
4	-2^{31} to $2^{31}-1$	$\pm 2.147 \times 10^{9}$
5	-2^{39} to $2^{39}-1$	$\pm 5.497 \times 10^{11}$
6	-2^{47} to $2^{47}-1$	$\pm 1.407 \times 10^{14}$
7	-2^{55} to $2^{55}-1$	$\pm 3.602 \times 10^{16}$
8	-2^{63} to $2^{63} -1$	$\pm 9.223 \times 10^{18}$

Table 4.2.1 - Capacities of Fixed Point Fields

4.2.1 Data Formats

The interpreter executes an intermediate code program from a 64K byte address space. Such a program contains four different types of data: signed fixed point numbers; pointers; byte strings and numeric strings. The interpreter can also interface with machine code programs and, for this purpose alone, employs machine addresses.

4.2.1.1 Signed Fixed Point Numbers

A signed fixed point number is between 1 and 8 bytes in length and resides in contiguous main storage. The byte at the low address is the most significant and its leftmost bit is interpreted as the sign bit. Positive numbers are represented in true binary notation with a sign bit set to zero. Negative numbers are held in two's-complement form with the sign bit set to one. The maximum positive number which can be represented is $2^{63} - 1$, and the maximum negative number is -2^{63}. The range of representation is therefore approximately $\pm 9.2 \times 10^{18}$. The capacity of the different lengths of numbers is given in Table 4.2.1.

Fixed point scaling is supported and a number can have up to 7 decimal places and a maximum of 18 significant digits. The scaling in force is indicated by the qualifier associated with each arithmetic instruction.

4.2.1.2 Pointers
A pointer is a two-byte quantity representing a byte location within the user program. It contains a value between 0 and 64K-1 represented in true binary notation. The byte at the low address is the most significant and its leftmost bit is **not** interpreted as a sign bit but is considered to represent the 32K unit position.

Any arithmetic performed on pointers by the interpreter is always modulus 64K, carries out of the senior bit position being ignored (i.e. the address space is circular, with location zero following location 64K-1).

The pointers need not be mapped directly onto a single area of memory. In particular the high locations, which will be used for the monitor program and system data areas, will normally be mapped into an area immediately preceding location zero. Hence, for example, location 64K-2 would normally be referred to as location -2.

4.2.1.3 Byte Strings
A byte string is a contiguous group of bytes which can be moved to or compared with another byte string. The interpreter is not concerned with the character coding employed except that in Move and Compare operations the interpreter will supply rightmost ASCII blanks to pad out short strings.

(The ASCII coding used by the interpreter is 8-bit with the senior, parity, bit set to zero.)

4.2.1.4 Numeric Strings
The general form of a numeric string is:

```
x y .z
```

where x is an ASCII plus or minus, y is a string of one or more ASCII digits making up the integral part of the number represented, and .z is an ASCII full stop or comma followed by one or more ASCII digits forming the decimal fraction. Any number of blanks can precede and follow the number.

Each of x, y and .z is optional. An unsigned quantity is treated as positive. One of the y or .z substrings must be present in any numeric string.

Note:

```
+1   VALID
1.1  VALID
1,1  VALID
.1   VALID
1.   INVALID
-01  VALID
+    INVALID
```

The numeric string instructions enable the program to convert numeric strings to signed fixed point numbers and vice versa.

4.2.1.5 Machine Addresses
The format of machine addresses is obviously dependent on the particular computer for which the interpreter is constructed. It is

assumed, however, that a particular computer will be capable of providing a **parameter address register** from one or more hardware components. This register must be capable of holding either a machine address or a pointer. It is used in starting interpretation and in passing parameters between the interpreter and machine code service routines.

4.2.2 Instruction Formats and Addresses

Each intermediate code instruction must begin on an even byte boundary. Its first five bits make up the operation code. This is followed by two address mode bits and the trap bit, used for program debugging. Except for short immediate mode instructions, the operation code byte is followed by a one byte qualifier whose usage differs from instruction to instruction. The content of the rest of the instruction depends on the address mode as shown in Figure 4.2.2.

4.2.2.1 Indexing

Every instruction has associated with it an initial effective address (IEA) which is the location of its initial target. The initial target is the true target only if the I register contains 0, signifying that indexing does not apply, or if the instruction is one of five (Load Length, Load Qualifier, Escape, Resume and Trace) which do not use indexing, or if it is a short immediate mode instruction.

When indexing applies the I register contains a positive integer in the range 1 to 32,767 inclusive. This quantity is decremented by one and then multiplied by the length of the target. The result is then added to the IEA to obtain the effective address of the target. During this calculation the arithmetic is modulus 64K.

Whenever the index register is used by an instruction it is reset to zero. An instruction, Load Index, is provided to allow the program to set the I register: this instruction may itself be indexed if it is preceded by another Load Index instruction. Resetting of the I register means that the indexing established by Load Index only applies to the instruction which immediately follows it, with the proviso that a Load Length, Load Qualifier or Trace instruction may be interposed between the Load Index and the subject instruction.

If the length register, L, contains a non-zero value this is the length to be used in the indexing calculation. Whenever the value in the L register is used it is reset to zero. An instruction, Load Length, is provided to allow the program to set the L register: this instruction cannot be indexed hence it can be generated following a Load Index instruction but before the instruction to be indexed. Otherwise if the length register is zero then the target length to be used is deduced from the operation code and, in most cases, the qualifier of the subject instruction.

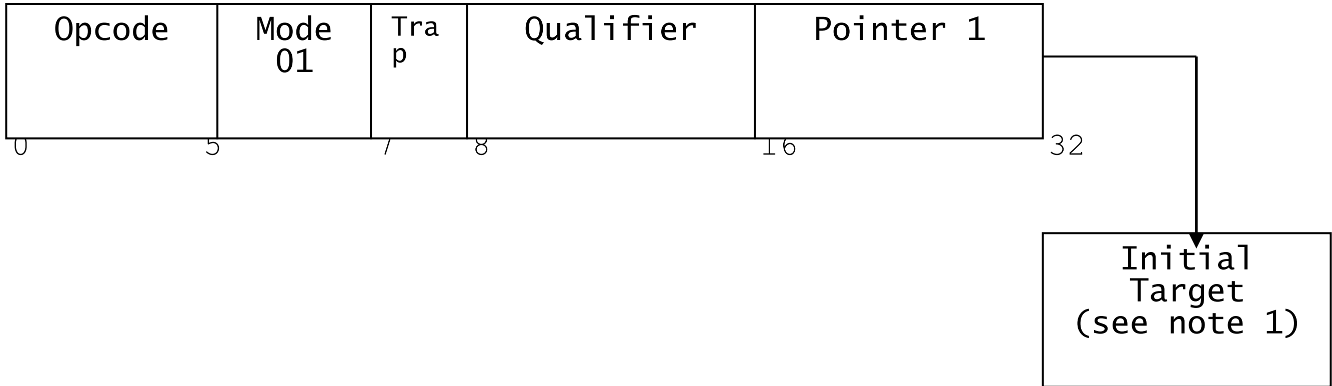
4.2.2.2 Immediate Instruction (MODE = 00)

In immediate instructions, which are mainly used for handling literals, the initial target immediately follows the instruction qualifier. The next sequential instruction, when it is defined, begins at the first even byte following the initial target.

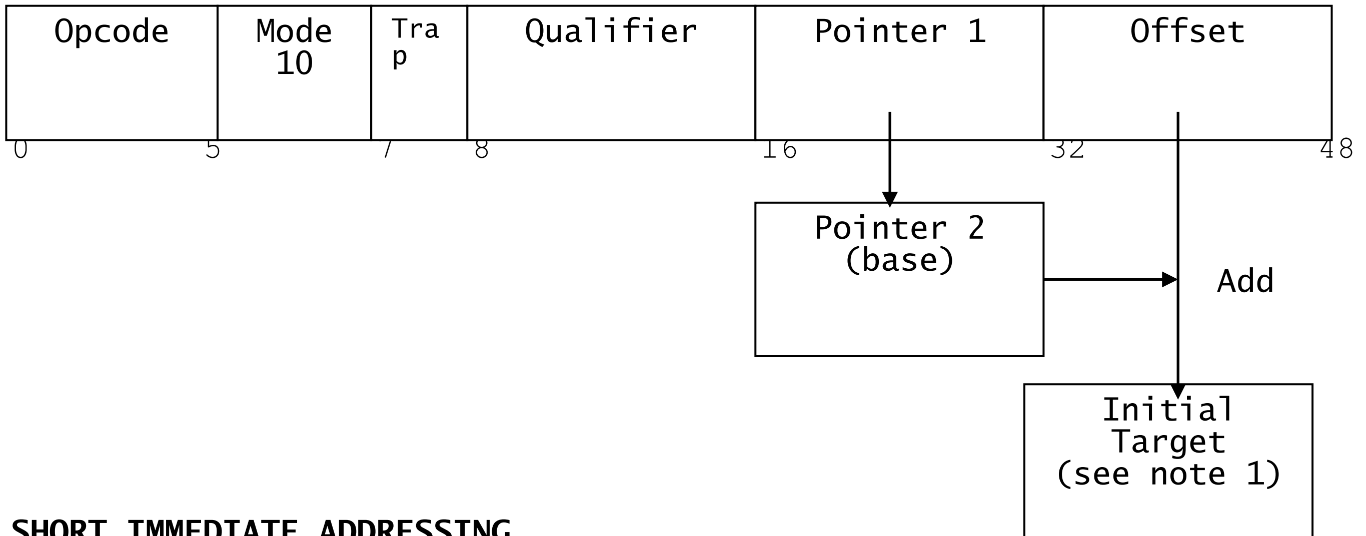
The Resume and Escape instructions have a 'null' target, i.e. one which is zero bytes in length (so has Pop List, when its qualifier is zero). Any such instruction is set up as an immediate instruction and the interpreter recognises from the operation code and qualifier that the target is zero bytes in length.

IMMEDIATE ADDRESSING

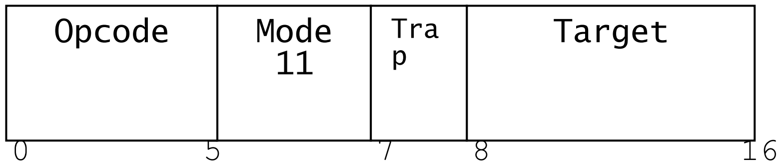
DIRECT ADDRESSING



BASED ADDRESSING



SHORT IMMEDIATE ADDRESSING



Note-1 If the Index Register, I, is non-zero the quantity (I-1) x length will be added to the initial Target Location to determine the true Target Location.

Figure 4.2.2 - Instruction formats and addressing

4.2.2.3 Direct Instructions (MODE = 01)
In direct instructions the third and fourth bytes are a pointer to the initial target, and the next sequential instruction follows this pointer.

4.2.2.4 Based Instructions (MODE = 10)

The based addressing mode is provided to allow Global Cobol access to data items within a linkage section and to support based variables and list processing. The instruction qualifier is followed by pointer 1 and a two-byte offset. Pointer 1 itself addresses a second pointer, the base. When the offset is added to the base the result is a pointer to the initial target. The next sequential instruction, when defined, begins at the byte following the offset.

4.2.2.5 Short Immediate Instructions (MODE = 11)

The short immediate address mode provides a very compact instruction format for single byte literals, which are quite common in Global Cobol programs. The second byte of the instruction is used as the initial target, rather than as a qualifier. The qualifier is always assumed to be 1. The next sequential instruction follows the second byte of the instruction.

System Area
Exception Number
Completion Code
Pointer to Start Instruction
Pointer to diagnostic logout area
Pointer to partition-0
Reserved (pointer to register area)
Pointer to IP flag

Diagnostic Logout Area	
Exception Number	
Completion Code	
Pointer to Failing Instruction	
A mill	
A scaling	A rounding
F register	Privilege status register
I register	
Q register	
T pointer	
T length	
Parameter stack index	
Seven 2-byte parameter stack entries	
Link stack index	
Twenty eight 4-byte link stack entries	
Pointer to last successful transfer of control instruction	
L register	

Machine Code Interface Area
Completion Code

Program Base Address
Exit Address
Stop Address
Number of entries, n, on Parameter Stack
The first n Parameter Stack entries converted to machine addresses
Machine code routine temporary work space. The size of this area is machine dependent.

Figure 4.2.3 - Areas shared by the user program and the interpreter

4.2.3 Registers and Data Areas

The interpreter can be seen as the CPU of an intermediate code virtual computer that uses registers and data areas which are initialised and updated by the instructions it obeys. Most of these registers and data areas can only be accessed by the interpreter and are described in sections 4.2.3.1 to 4.2.3.9 below. The final four areas described (4.2.3.10, 11, 12, 13) may be accessed by the program as well as by the interpreter.

4.2.3.1 The Arithmetic Accumulator (A)

To support scaled arithmetic the arithmetic accumulator contains three sub-accumulators:

- The **mill**, an 8-byte register in which integer arithmetic takes place;
- The **scaling factor**, a one-byte register whose value indicates the number of decimal places in the integer contained in the mill;
- The **rounding flag**, used to support rounding following division. It is set to zero if the absolute value of the remainder is less than half the absolute value of the divisor, and otherwise to +1 or -1 according to whether the divisor and dividend have the same or opposite signs.

4.2.3.2 The Flag Register (F)

The F register is a 1-byte register which can assume only the values 0 and 1. It is used in tentative exception handling. The register is set to 1 when a tentative exception occurs and reset to 0 by Stop, Call, Exit and Jump instructions which test the F condition and clear the pending tentative exception. It is also cleared when a tentative exception becomes an immediate exception.

4.2.3.3 The Privilege Status Register

The Privilege Status register is a 1-byte register which indicates whether the code currently being executed is privileged or unprivileged. This is used in connection with interrupts (4.2.3.11) and memory protection (4.2.6).

The register may have the following values:

- -1 means that the code currently being executed is unprivileged;
- zero or a positive number means that the code currently being executed is privileged, and the value is the level of the link stack at the time the privileged status was enabled.

The register is set by the Pop List instruction, and may be reset to -1 by an Exit instruction.

4.2.3.4 The Index Register (I)

The Index register can be considered as a two-byte numeric variable containing an integer between 0 and 32,767 inclusive. When the register is non-zero it is used in indexed address calculations as explained in 4.2.2.1. The Load Index instruction is used to establish the register's contents which are reset to zero once it has been used.

4.2.3.5 The Length Register (L)

The Length register can be considered as a two-byte numeric variable containing an integer between 0 and 32,767 inclusive. When the register is non-zero it contains the length to be used in the next index address calculation as explained in 4.2.2.1. The Load Length instruction is used to establish the register's contents which are reset to zero once it has been used.

4.2.3.6 The Qualifier Register (Q)

The Qualifier register can be considered as a two-byte numeric variable containing an integer between 0 and 65,535 inclusive. When the register is non-zero it is used in place of the instruction's qualifier. The Load Qualifier instruction is used to establish the register's contents which are reset to zero once it has been used.

4.2.3.7 The Text Accumulator (T)

The Text registers consist of two 2-byte fields: the pointer field, which locates the T string, and the length field, which contains the string size in bytes (0 - 65,535). T register contents are established by the Set and Call Display instructions and used in the Move, Compare and Exchange instructions.

4.2.3.8 The Link Stack

The link stack consists of a two-byte index field containing a value between 1 and 28, and 28 four-byte link stack entries. The index field indicates the number of outstanding Call statements. If non-zero it indexes the link stack entry used for the previous Call.

Each link stack entry contains two pointers. The first is a two-byte pointer to the next sequential instruction following the Call. The second is a pointer to the instruction to which Call passed control and is provided to facilitate debugging.

The link stack is maintained by the Call and Exit instructions, Exit being responsible for popping entries from the stack.

4.2.3.9 The Parameter Stack

The parameter stack consists of a two-byte index field, containing a value between 0 and 7, and 7 two-byte parameter stack entries.

The index field indicates the number of entries placed on the stack by outstanding Push instructions. Each entry is a pointer to the data of the corresponding Push.

The stack is maintained by Push, Pop, Pop List and Escape instructions and is provided to support the passing of parameters via the Global Cobol USING clause and the implementation of based variables.

4.2.3.10 The System Area

The two bytes at locations -2 and -1 (see 4.2.4) contain a pointer to the system area, which can be accessed by both the interpreter and the user program.

As Figure 4.2.3 shows, the system area is subdivided into 7 two-byte fields. These are briefly discussed here for the sake of completeness, but are dealt with in more detail in the sections describing the features which use them.

The **exception number** is set to zero when interpretation starts and is set to a non-zero identifying value when a program exception occurs.

The **completion code** is set to the two-byte target of a successful Exit or Stop instruction.

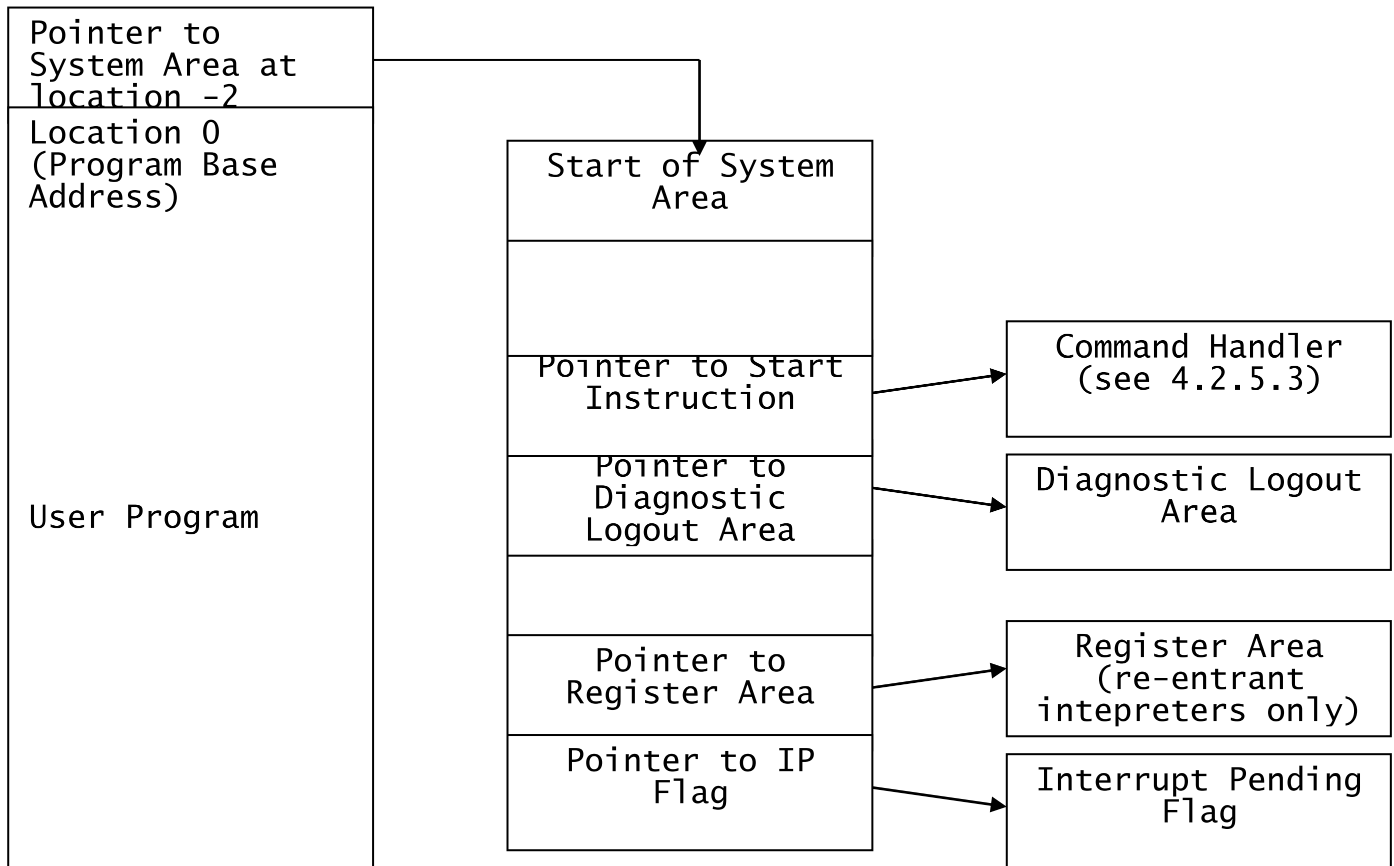
The **pointer to start instruction** locates the intermediate code instruction in the monitor program which is given control when interpretation starts and also when an immediate program exception occurs.

The **pointer to the diagnostic logout area** supplies the interpreter with the location of a 160-byte area in which the interpreter preserves its status when an immediate program exception takes place.

The **pointer to partition-0** is set by the interpreter to point at **address 0** of the partition in which the Global Cobol program is executing. In single partition systems this is the same as absolute location 0.

The **pointer to the IP flag** is maintained by the Global Cobol user program. The use of the flag is described below.

Figure 4.2.3.10 shows how these areas are linked.



4.2.3.10 - Addressing system data areas

upt-Pending Flag (IP)
The interrupt-pending flag is set in order to 'interrupt' the intermediate code computer; for example, in order to halt a program which is in a loop so that it can be debugged.

The flag is a single byte which is zero if not set, and contains the exception number to be returned if set. It is addressed by the pointer to the IP flag in the system area. Note that this pointer may be changed at any time by the program which is being executed.

The flag is tested by transfer of control instructions, provided that the Flag register is zero (i.e. no tentative exception is outstanding) and the privilege status register is negative (i.e. the instruction is unprivileged) and the previous instruction was not a Resume. If the flag is non-zero the exception number given by the flag will be returned and the flag reset to zero.

Exception number 1 is used to cause a 'break' exception, and exception numbers 20-29 are reserved for Global use as interrupt exceptions. Global Cobol is responsible for the assignment of these exception numbers.

4.2.3.12 The Diagnostic Logout Area
The diagnostic logout area is set up by the interpreter whenever an immediate program exception takes place. The exception number and completion code are copied from the system area. The next field (see Figure 4.2.3) is a pointer to the instruction which failed; there follow the contents of the registers and stacks at the point of failure.

The penultimate field in the diagnostic logout area is a pointer to the last transfer of control instruction which caused a transfer of control. This is to facilitate debugging a program which transfers control to a location which does not contain a valid instruction.

4.2.3.13 The Machine Code Interface Area

The machine code interface area is set up by the interpreter when it honours an Escape instruction. The address of the area is passed to the machine instruction following the Escape in the parameter address register. On any computer with 16-bit machine addresses the area will be as defined in Figure 4.2.3.

The completion code field is set to zero by the interpreter, but may be updated by the machine code routine.

The program base address allows the routine to convert pointers to machine addresses and vice versa.

The next two addresses are the location of routines within the interpreter (see the Escape instruction for usage details).

The next field is the number of entries on the parameter stack when Escape occurred. This is followed by the active parameter stack entries converted from pointer to machine address format.

4.2.3.14 The Interpreter Initial State

When interpretation first begins or when an immediate program exception occurs, the following interpreter registers are set to zero to establish the interpreter initial state:

- A mill, scaling factor and rounding flag;
- F;
- I;
- L;
- Q;
- T pointer and length;
- Link stack index;
- Parameter stack index;
- Privilege status.

Error number	**Explanation**
0	An Exit instruction has been attempted when there is no outstanding Call in the link stack. This is not a true error: issuing an Exit from the highest level routine is a convenient means of ending interpretation.

1	The machine address passed in the parameter address register to identify location zero at the start of interpretation was invalid.
2	Not used.
3	The start address pointer in the system area which should point at the instruction to be executed at start of interpretation contains an odd value. This error can occur either at start of interpretation or when an immediate program exception occurs.
4, 5, 6 etc.	An internal error has arisen in the interpreter. Assignment of error numbers is implementation dependent.

Figure 4.2.4 - Terminal error number assignments

4.2.4 Starting and Ending Interpretation

To start interpretation a machine code program known as the steering routine must pass control to the entry point of the interpreter supplying, in the parameter address register, the machine address of location zero of the intermediate code program, the program base. Some implementations may insist that this machine address is even. All implementations, however, allow this address to be established at run-time. This means that Global Cobol code is **dynamically** relocatable. All programs can be translated and linkage edited relative to location zero; the actual run-time start address is immaterial.

The facility is similar in concept to the program base register mechanism employed in some models of the ICL 1900 series. For clarity we always refer to **locations** within the user program, starting at 0, as distinct from **machine addresses** within the address space of the computer itself.

When entered the interpreter sets its internal registers to the initial state defined in 4.2.3.13 and zeroises the exception number in the system area. Interpretation begins with the start instruction of the monitor routine, given by the pointer in the system area. Typically this routine would load a user program and then enter it via a Call instruction.

The interpreter returns control to the steering routine either when interpretation is finished or if a terminal error occurs. The parameter address register is used to supply a **terminal error number**, 0 signifying normal completion. Figure 4.2.4 lists the error numbers and their meanings.

4.2.5 Program Exception Handling

In Global Cobol documentation an intermediate code program exception is always termed a program check. The so-called 'exceptions' returned by Global Cobol routines result from the execution of the Exit instruction with a non-zero target, as expanded by the EXIT WITH CODE statement. Such an Exit instruction generates a pending tentative exception. This exception will be cleared when the next Global Cobol statement to be executed is ON EXCEPTION or ON OVERFLOW because the first instruction such a statement generates is a Jump conditional on the F register.

When a program exception occurs the interpreter registers are saved in the logout area and then execution continues from the start instruction of the monitor program. Such an exception is termed an **immediate exception**.

In addition the interpreter allows the application program to handle certain exception conditions itself by means of the technique of **tentative exceptions.**

4.2.5.1 Tentative Exception Handling

When certain exception conditions arise (such as numeric conversion - there are others) the exception number is stored in the normal way but instead of a program exception taking place immediately all that happens is that the F register is set on (i.e. assigned the value 1). This signifies that a tentative exception is pending.

Certain instructions are suppressed if a tentative exception is pending: these are Move, Store, Round, Divide, Decimal, Load Index, Load Qualifier, Load Length, Exchange and Store Unsigned. These instructions are treated as no-operations if an exception is pending and the next sequential instruction is executed.

Status switching instructions (Resume, Escape and Trace) may be executed while an exception is pending. Trace will not affect the pending exception; Resume resets the F register; Escape will not affect the exception unless it completes with a Stop or Exit exception which will cause the pending exception to be made immediate.

A tentative exception is cleared by any transfer of control instruction conditional on the F register. 'Clearing' means turning the F register off.

Execution of any other instruction, including transfers of control not conditional on the F register, will cause the tentative exception to be made immediate (see 4.2.5.2). The effect is that if a program expects a tentative exception and contains the logic to handle it, it will be allowed to continue. Otherwise the F register will not be tested in time; an instruction which cannot be suppressed will be executed and the exception made immediate.

The reason for suppressing certain instructions rather than making them cause exceptions is to simplify the generation of intermediate code from Global Cobol such as:

```
ADD A TO B GIVING C
ON OVERFLOW GO TO HANDLER
```

This expands (conceptually) to:

```
Load A          * cannot cause tentative exception
Add B                * may cause tentative exception
Store C         * may be suppressed, may cause tentative exception
Jump F, HANDLER      * goes to HANDLER clearing pending exception
if set.
```

Note that if another exception occurs while a tentative exception is pending then the tentative exception is made immediate and the second exception is ignored.

4.2.5.2 Immediate Exception Handling

An immediate exception can occur in three situations. The first case is when a tentative exception is outstanding, and the program attempts to execute an instruction which is not either suppressible or a

transfer of control instruction conditional on the flag register. In this case the tentative exception becomes an immediate exception.

The second case is if any further exception occurs while a tentative exception is outstanding, in which case the original tentative exception again is returned as an immediate exception. This most commonly occurs if one of the instructions executed while the exception is outstanding has its trap bit set.

The third case is if there is no tentative exception outstanding, and an exception which cannot be a tentative exception occurs, such as Trap or Interrupt.

Once the exception number has been determined status and other information is preserved in the diagnostic logout area as follows:

- the exception number and completion code are taken from the system area;

- the pointer to the failing instruction is set to address the start of the instruction which returned the exception, except in the case of an exit exception when it addresses the instruction to which Exit returned control, i.e. the instruction which failed to test for the pending exception;

- the current values of the interpreter registers and stacks are saved. The value of the flag register is always zero. For exceptions other than Trap the values of the registers may have been corrupted by the processing of the failing instruction;

- the pointer to the last successful transfer of control instruction is set to point to the start of the last instruction executed which caused a transfer of control to take place.

The interpreter registers and stacks are then returned to their initial values as described in 4.2.3.14. Finally the interpreter executes the Start instruction using the pointer in the system area.

4.2.5.3 Global Considerations

The intermediate code routine whose first instruction is given by the Start instruction pointer in the system area is part of the Global System Manager monitor program. It is entered when interpretation first begins and whenever an immediate program exception takes place.

On entry to the routine the interpreter registers are set to the initial values defined in 4.2.3.14. The exception number in the system area will be zero when interpretation first begins or an integer in the range 1 to 17 if the command handler has been invoked to service an exception. The routine's first action is therefore to examine the exception number to determine why it has been entered.

In the case of the Stop exception (number 5) the completion code in the system area specifies additional information which the routine can process. This allows the routine to assume the role of a job scheduler. For example the Global Cobol:

```
    CHAIN    program-id
```

returns completion code -1 to indicate that chaining is to occur, the name of the chained program having been set up in a system variable.

The Global monitor is responsible for assigning and processing completion codes.

4.2.5.4 Exception Number Assignments

Table 4.2.5.4 lists and describes the exceptions which can be generated by the interpreter. Some of these exceptions are discussed in more detail below.

Exception 1 (Break) and exceptions 20-29 occur as a result of a program interrupt, as described in 4.2.3.11.

A trap exception occurs if an instruction is executed which has its trap bit set on. This feature is for use when debugging in order to cause interpretation of a program to be halted at a particular point so that registers and data areas may be examined. The program may be resumed using the Resume instruction. The trap exception cannot occur on the first instruction following a Resume, hence if the trap bit is left set following a trap exception the instruction will be executed when the program is resumed, but will cause a trap exception if executed subsequently.

The Stop exception is used to indicate to the monitor program that a Stop instruction has been executed.

The Exit code exception is part of the mechanism which allows a subroutine to cause a tentative exception if it detects an error (see Exit for details).

Exception number	Name	Explanation
1	Break	A transfer of control instruction was executed while the interrupt-pending flag was set to 1.
2	Illegal Address	No longer used.
3	Illegal Load	The target of the failing Load Length or Load Index instruction had an integral part which was negative or zero. Under V5.0, also returned for Load Qualifier.
4	Trap	The failing instruction has its trap bit set. It has not been executed.
5	Stop	The failing instruction is a Stop instruction which has succeeded. This exception is used in the implementation of the Global Cobol STOP RUN statement.
6	Illegal Operation	The failing instruction had an operation code which is not assigned, or is not valid with this addressing mode.
7	Illegal Qualifier	The qualifier associated with the failing instruction did not have a value acceptable to that class of instruction.
8	Illegal Branch	The target of a Call or Jump instruction contains an odd value.
9*	Exit Code	An Exit instruction has succeeded, setting a non-zero completion code. This condition is always treated as a tentative exception.

10*	Numeric Conversion	The target of a Binary instruction was not a valid numeric string.
11*	Arithmetic Overflow	An overflow condition has prevented an Add, Subtract, Multiply,Divide, Store, Round, Binary or Decimal instruction from completing. The condition is always treated as a tentative exception.
12	Memory Violation	An unprivileged Decimal, Move, Exchange, Store, Store Unsigned or arithmetic instruction with the store option has attempted to write to a location below the start of the system area, or above the top of the user area.
13*	Parameter Stack	A Push or Call Display instruction has attempted to place an eighth item on the parameter stack; a Pop instruction has been attempted when the stack is empty; a Pop List instruction has failed because the number of entries in the list was not equal to the number of parameters on the stack. This condition is always treated as a tentative exception.
14	Link Stack	A Call or Call Display instruction has succeeded when there are already 28 Calls outstanding and the link stack is full.
15	Advance	Used by Advance instruction in $DEBUG.
16	No Base	The base associated with a Based Mode instruction (other than Push) contains the value #FFFF (=uninitialised pointer) and the instruction is not privileged.
17	Scaling	A Store Unsigned or And instruction specifies a different number of decimal places to the value in the accumulator, and hence would require scaling. Some interpreters do not detect this exception.
18-19		Reserved for future use.
20-29		Interrupt exceptions.

* Tentative Exceptions

Table 4.2.5.4 - Exception number assignments

4.2.6 Privileged Code and Memory Protection

The memory protection feature is provided so that ordinary programs can be prevented from accidentally corrupting the Global Cobol monitor. Monitor routines, which are allowed to access fields within the monitor, run in a special privileged mode. In addition privileged routines cannot suffer an interrupt exception: this allows critical processing in the monitor and system routines to be protected against interruption.

The status of the instruction being executed is given by the Privilege Status register, which can have the values:

- -1, meaning that the code currently being executed is unprivileged;
- zero or a positive number, meaning that the code currently being executed is privileged. The value is the level of the link stack at the time the privileged status was enabled.

The privileged mode of operation is enabled and disabled by the Pop List instruction. The senior bit of the qualifier (or the senior bit of the junior byte of the Q register if set) is a flag which is zero if the Pop List is unprivileged and 1 if privileged. Privileged mode can also can be changed to unprivileged by an Exit instruction.

Initially, and following a program check, the privileged status will be set to zero, so that the command handler is privileged.

If the code currently being executed is privileged then if an unprivileged Pop List instruction is encountered the status is changed to -1 (unprivileged).

When an Exit instruction is executed, if the resulting link stack index is less than the privileged status register then the status is changed to -1 (unprivileged).

The instructions Decimal, Move, Exchange, Store and Store Unsigned, and arithmetic instructions with the store option will, if the privileged status is -1, check for memory violation and return exception 12 if detected. The basic test for memory violation is that the target address is greater than or equal to the start of the system area. Some intermediate code interpreters may perform more elaborate tests, for example to protect areas of memory containing parts of the machine's operating system.

4.2.7 Intermediate Code Instructions

Table 4.2.7 lists the 29 intermediate code instructions honoured by the interpreter. Note that there are no I/O instructions as all I/O is performed by service routines: the Escape instruction allows Call to invoke machine code routines as well as Global Cobol routines.

The notes in the table are as follows:

A. Transfer of Control instructions which are conditional on the F register can execute following a tentative exception and clear the tentative exception;

B. These operations are executed as no-operations when a tentative exception is outstanding;

C. The Resume, Escape, Trace Load Qualifier and Load Length instructions cannot be indexed.

These instructions are described in detail in the following chapters.

Instruction class	Instruction name	Title	Opcode	Notes
Byte String	Set	Set Byte String attributes	0	
	Move	Move Byte String	1	B
	Compare	Compare Byte Strings	2	
	Exchange	Exchange Byte Strings	26	B
Numeric String	Binary	Convert to Binary	3	
	Decimal	Convert to Decimal	4	B
Arithmetic	Load	Load Accumulator	5	

	Store	Store Accumulator	6	B
	Round	Round Accumulator	7	B
	Add	Add to Accumulator	8	
	Subtract	Subtract from Accumulator	9	
	Multiply	Multiply Accumulator	10	
	Divide	Divide into Accumulator	11	B
	Store Unsigned	Store Unsigned Accumulator	27	B
	And	AND Accumulator	28	
Transfer of Control	Jump	Jump Conditionally	12	A
	Call	Call Section Conditionally	13	A
	Stop	Stop Run Conditionally	14	A
	Exit	Exit Conditionally	15	A
Status Switching	Resume	Resume Failed Program	16	C
	Escape	Escape to Machine Code	17	C
	Trace	Trace Identifier	18	C
Parameter Stack	Push	Push Pointer onto Stack	19	
	Pop	Pop Pointer from Stack	20	
	Pop List	Pop Parameter List from Stack	21	
Register Load	Load Index	Load Index Register	22	B
	Load Qualifier	Load Qualifier Register	23	B,C
	Load Length	Load Length Register	25	B,C
Special Purpose	Call Display	Call Display SVC	24	

Table 4.2.7 - Instruction list

4.3 Byte String Instructions

This chapters explains the instructions used in intermediate code to manipulate byte strings. The Set instruction loads the T registers which give the source address for Move and Compare instructions. Move performs a byte string move, Compare a byte string comparison. Exchange interchanges two strings. By using the Q register up to 65,535 bytes can be moved, exchanged or compared with one instruction.

4.3.1 Set - Set T Registers

Set establishes the T length and T pointer registers. T pointer addresses the target of Set. T length is taken from the qualifier of Set (or from the Q register if the previous instruction was Set Qualifier). The length can therefore be between 0 and 32,767 bytes.

4.3.1.1 Exceptions

The following exceptions may occur:

16 No Base

4.3.1.2 Registers or Storage Modified

The following registers and storage are modified:

T pointer
T length
I, L and Q are cleared

4.3.1.3 Programming Notes
It is possible to use a zero qualifier to set up a null string.

4.3.2 Move - Move Byte String
Move transfers the byte string addressed by T pointer to its target. The length involved in the transfer is the qualifier of Move (or the Q register). Truncation occurs if this length is smaller than T length. If, however, the T string is the smaller, then rightmost ASCII blanks are inserted during the transfer.

4.3.2.1 Exceptions
The following exceptions may occur:

12 Memory Violation
16 No Base

4.3.2.2 Registers or Storage Modified
The following registers and storage are modified:

I, L and Q are cleared

4.3.2.3 Programming Notes
The move takes place one byte at a time from left to right. Hence this instruction may be used to propagate a value through a string.

4.3.3 Compare - Compare Byte Strings
Compare sets the A accumulator to 0, -1, or +1 according to whether the byte string at its target is equal to, greater than or less than the T string. The length used in the comparison is the greater of the T length and the instruction qualifier (or Q if set). The rounding flag is cleared.

Comparison proceeds a byte at a time in ascending address order. If the strings are not of equal length the smaller is padded with rightmost ASCII blanks. Comparison stops as soon as two different bytes are met, in which case the string containing the byte with the highest binary value is the greater. If the end of the strings is met and there is no difference, then equality is returned.

4.3.3.1 Exceptions
The following exceptions may occur:

16 No Base

4.3.3.2 Registers and Storage Modified
The following registers and storage are modified:

A accumulator
I, L and Q are cleared

4.3.3.3 Programming Notes
The instruction qualifier may be zero, in which case the T string will be checked for being all ASCII spaces.

4.3.4 Exchange - Exchange Byte Strings
Exchange interchanges the byte string addressed by the T pointer and its target string. The length involved in the exchange is given by the T length register - the length specified by the qualifier of the exchange instruction (or the Q register) is ignored.

4.3.4.1 Exceptions
The following exceptions may occur:

12 Memory Violation
16 No Base

4.3.4.2 Registers or Storage Modified
The following registers and storage are modified:

Storage areas addressed by T pointer and the instruction
I, L and Q registers cleared

4.3.4.3 Programming Notes
The effect of an exchange instruction on two overlapping areas is not defined. On most processors the instruction executes faster if the fields are even byte aligned.

4.4 Numeric String Instructions
The Decimal instruction converts the contents of the A accumulator into a standard format numeric string at the target. There are no trailing blanks.

The Binary instruction converts the numeric string at the target into a binary value which is placed in the mill of the A accumulator. The scaling factor for the A accumulator is deduced from the instruction qualifier.

The format of the qualifier is given in Figure 4.4. The number of digits preceding the decimal point must be at least 1, otherwise an illegal qualifier exception will occur. Also, if the number of digits (p + q) exceeds 19 an illegal qualifier exception will occur.

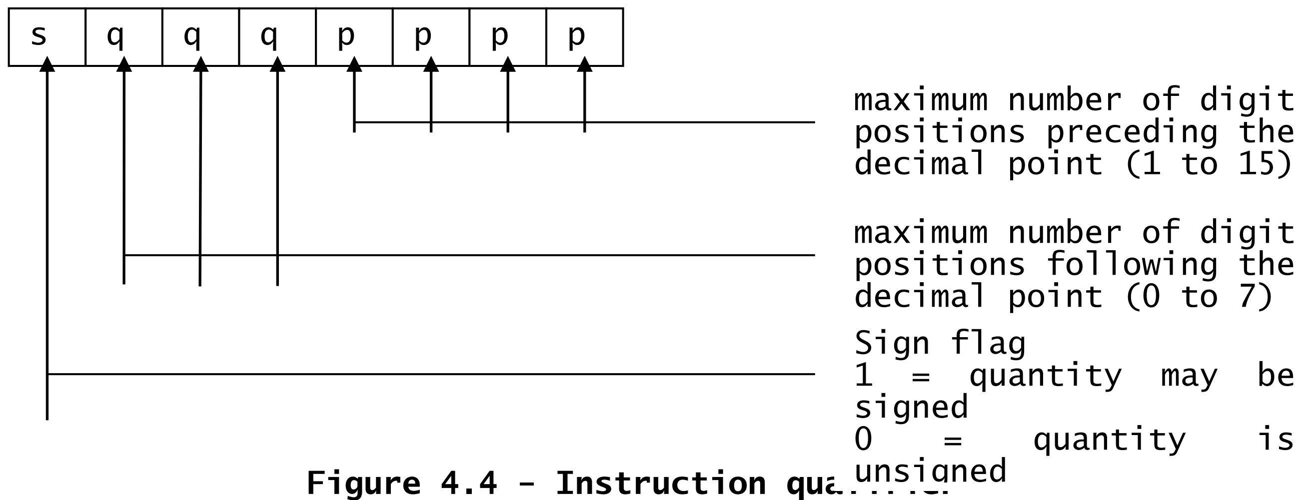
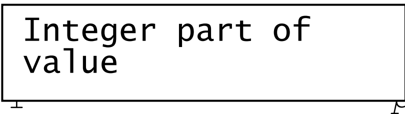
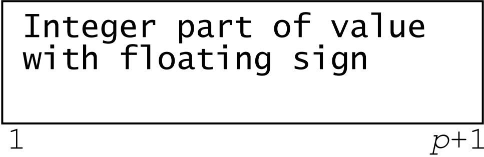


Figure 4.4 - Instruction qua...

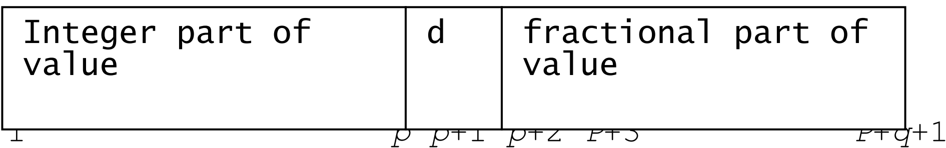
s=0, q=0 Length is p characters



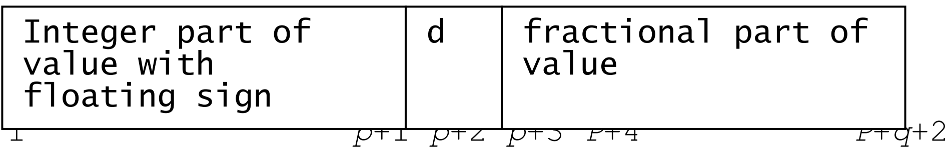
s=1, q=0 Length is p+1 characters



s=0, q>0 Length is p+q+1 characters



s=1, q>0 Length is p+q+2 characters



Notes
The integer part has leading zeros suppressed but at least one digit is always printed (i.e. zero prints as a single '0'). If the fractional part of the value has too few digits, then the remaining digits are shown as zeros.

d represents the decimal point character (a period or a comma) which is held in a field in the system area.

Figure 4.4.1 - Format of string produced by decimal instruction

4.4.1 Decimal - Convert Accumulator to Numeric String

The Decimal instruction converts the value in the accumulator to a standard format numeric string in accordance with the instruction qualifier. The string is stored in the target. The value of the accumulator is not changed.

The format of the string produced is given by Figure 4.4.1. Note that at least one digit must precede the decimal point.

Overflow will occur if the value in the accumulator is too large for the format specified or if it is negative and an unsigned format is specified. Overflow will also occur if the scaling factor of the accumulator is not in the range -7 to +15.

4.4.1.1 Exceptions

The following exceptions may occur:

7	Illegal Qualifier
11	Overflow
12	Memory Violation

16 No Base

4.4.1.2 Registers and Storage Modified
The following registers and storage are modified:

Target field in store
I, L and Q are cleared

4.4.2 Binary - Convert Numeric String To Binary
The Binary instruction converts a numeric string to a fixed point number and stores the result in the accumulator.

The scaling of the accumulator is the number of decimal places specified in the qualifier (q). The division rounding flag is cleared.

The length of the input string to be converted is determined from the qualifier as follows:

```
s=0, q=0  length is p
s=1, q=0  length is p+1
s=0, q>0  length is p+q+1
s=1, q>0  length is p+q+2
```

If the input string is not a valid numeric string (see 4.2.1.4) or has too many digits following the decimal point, then a numeric conversion exception will occur. If the string contains more than p digits (including leading zeros) before the decimal point, or if it is signed and the format is unsigned, then an overflow exception will occur. An overflow exception will also occur if the magnitude of the value input is greater than or equal to 2^{63} x 10^{-q}. Extra digits following the decimal point are ignored, and if fewer than q are supplied the remainder are set to zero. If q=0 the string must not contain a decimal point.

4.4.2.1 Exceptions
The following exceptions may occur:

7 Illegal Qualifier
10 Numeric Conversion
11 Overflow
16 No Base

4.4.2.2 Registers and Storage Modified
The following registers and storage are modified:

A accumulator
I, L and Q are cleared

4.4.2.3 Programming Notes
A value of spaces is not a valid numeric string and will cause a numeric conversion exception.

A '+' sign may be input even if the format is unsigned, although this reduces the number of digits which may be input in the number.

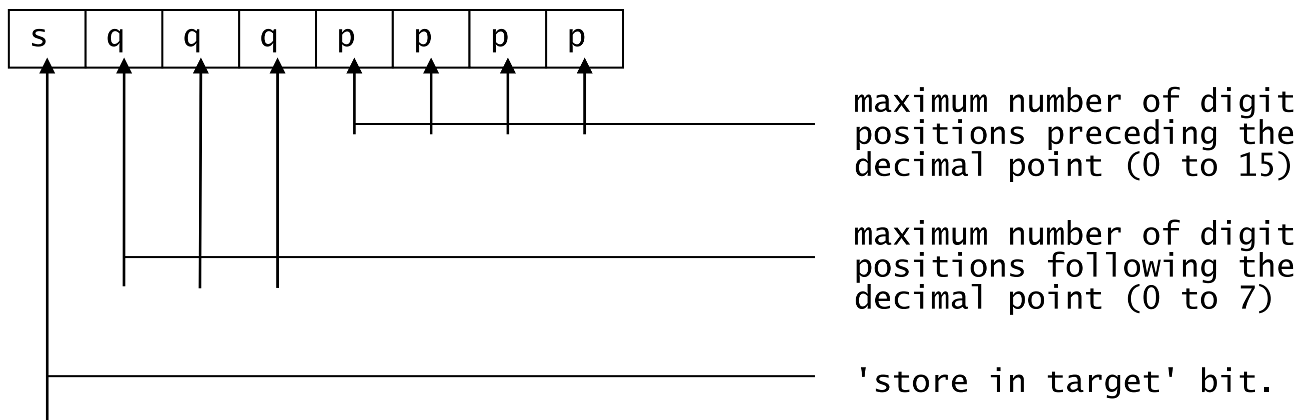
4.5 Arithmetic Instructions
The arithmetic instructions perform scaled fixed point arithmetic. Numbers can have a maximum of 18 significant digits with up to 15 digits before the decimal point and 7 after it. To reduce Global Cobol

code size the Round, Add, Subtract, Multiply, Divide and And instructions can optionally be requested to store their result in the target. Thus Global Cobol ADD A TO B can be implemented by a Load and Add rather than requiring a Load, Add and Store.

All the arithmetic instructions except Load, Store Unsigned and And can suffer arithmetic overflow. However, this is processed as a tentative exception to support the optional Global Cobol ON OVERFLOW statement which can follow arithmetic statements.

If overflow occurs on any arithmetic statement the target field will remain unchanged; in particular it will not be corrupted if overflow is detected during a store operation.

To control scaling and the optional storing of the result in the target the arithmetic instruction qualifier has a format somewhat similar to that used with numeric string instructions:



The s bit is ignored in the qualifier of Load, Store and Store Unsigned instructions.

Because the mill of the A accumulator is only 8 bytes in length a maximum of 19 significant digits can be supported. Therefore the sum of p and q must be no greater than 18 or an illegal qualifier immediate program exception will take place when the instruction is interpreted.

An illegal qualifier exception will also occur if p=q=0, since a fixed point number with no digits before or after the point is meaningless.

The Add, Subtract, Store and Round instructions require numbers to be **scaled** by division or multiplication by a power of ten so that the target field and the accumulator have the same precision. For example, if the A mill contains 1234 with an A scaling of 3, representing 1.234, and it is to be stored in a field with 1 decimal place then the A mill must be divided by 102 so that it contains 12 corresponding to scaling of 1 and representing 1.2. The scaling factor is always updated to reflect the new scaling of the accumulator.

4.5.1 Load - Load Accumulator

The target of the Load instruction is loaded into the A mill. The A scaling is set to the number of places following the decimal point (q) as given by the qualifier. The rounding flag is cleared.

4.5.1.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
16 No Base

4.5.1.2 Registers and Storage Modified
The following registers and storage are modified:

A accumulator
I, L and Q are cleared

4.5.2 Store - Store Accumulator
Store scales the accumulator to match the target field and stores it in the target field. If the value in the accumulator is too large for the target field overflow will occur (see Table 4.2.1.1). The rounding flag is cleared.

4.5.2.1 Exceptions
The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.2.2 Registers and Storage Modified
The following registers and storage are modified:

A accumulator
Target location in store
I, L and Q are cleared

4.5.2.3 Programming Notes
Note that the value in the A accumulator is modified by the Store instruction.

4.5.3 Round - Round Accumulator
Round scales the accumulator to match the target field and rounds the last digit of the result to the nearest digit.

If the A scaling factor is less than the scaling of the target (q) then A is scaled up but no rounding is performed.

If the A scaling factor is equal to the scaling of the target then the result is rounded away from zero if the rounding flag is set (this flag is set if the next digit of the quotient would be 5 or greater).

If the A scaling factor is greater than the scaling of the target then A is scaled down and if the most significant of the digits truncated during scaling was a 5 or greater the result is rounded away from zero.

The rounding flag is always cleared.

If the store in target option is specified then the accumulator is stored in the target field.

Overflow will occur if the scaling or rounding gives a result outside the range $-2^{63} \times 10^{-r}$ to $(2^{63}-1) \times 10^{-r}$, where r is the number of decimal places in the result. Overflow will also occur if the store option is

specified and the result is too large for the target field (see Table 4.2.1.1).

4.5.3.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.3.2 Registers and Storage Modified

The following registers and storage are modified:

A accumulator
Target field (if store option specified)
I, L and Q are cleared

4.5.3.3 Programming Notes

This instruction is normally used with the store option specified; if this option is not specified then the address of the target field is not used by the instruction.

4.5.4 Add - Add to Accumulator

Add adds the target number to the number in the accumulator and places the result in the accumulator.

The number of decimal places in the result, stored in A scaling, is the greater of q (in the qualifier) and the original A scaling (e.g. 2.1 added to .234 would give 2.334). The rounding flag is cleared.

If the store option is specified in the qualifier then the accumulator is scaled to match the target field and stored in the target field.

Overflow will occur if scaling or addition give a result which is not in the range $-2^{63} \times 10^{-r}$ to $2^{63}-1 \times 10^{r}$, where r is the number of decimal places in the result. Overflow will also occur if the store option is specified and the result is too large for the target field (see Table 4.2.1.1).

4.5.4.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.4.2 Registers and Storage Modified

The following registers and storage are modified:

A accumulator
Target field (if store option specified)
I, L and Q registers

4.5.5 Subtract - Subtract from Accumulator

Subtract subtracts the target number from the number in the accumulator and places the result in the accumulator.

The number of decimal places in the result, stored in A scaling, is the greater of q (in the qualifier) and the original A scaling (e.g. 2 minus .01 gives 1.99). The rounding flag is cleared.

If the store option is specified in the qualifier the accumulator is scaled to match the target field and stored in the target field.

Overflow will occur if the result is not in the range $-2^{63} \times 10^{-r}$ to $2^{63}-1 \times 10^{-r}$, where r is the number of decimal places in the result. Overflow will also occur if the store option was specified and the result is too large for the target field (see Table 4.2.1.1).

4.5.5.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.5.2 Registers and Storage Modified

The following registers and storage are modified:

A accumulator
Target field (if store option specified)
I, L and Q are cleared

4.5.6 Multiply - Multiply Accumulator

The value in the accumulator is multiplied by the value of the target number, and the result is placed in the accumulator.

The number of decimal places in the result, stored in A scaling, is the original value of A scaling plus the number of decimal places in the multiplier (q).

The rounding flag is cleared.

Overflow will occur if either operand has the value $-2^{63} \times 10^{-s}$ where s is its scaling (i.e. the largest negative value which may be held in 8 bytes). Overflow will also occur if scaling or subtraction give a result whose magnitude is greater than or equal to $2^{63} \times 10^{-r}$ where r is the scaling of the result, as defined above.

If the store option is specified in the qualifier then the accumulator is scaled to match the target and stored in the target. Overflow will occur if the result is too large for the target field (see Table 4.2.1.1).

4.5.6.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.6.2 Registers and Storage Modified

The following registers and storage are modified:

A accumulator
Target field (if store option specified)
I, L and Q are cleared

4.5.6.3 Programming Notes

Note that the precision of the result can be greater than 7 decimal places.

For some interpreters the time taken to execute a Multiply will depend on the magnitude of the target number and hence, where possible, the smaller of the two numbers should be used as the target of the Multiply instruction.

4.5.7 Divide - Divide into Accumulator

Divide divides the value in the A accumulator by the value of the target number, and places the result in the accumulator. The result is positive if both operands have the same sign, negative if their signs differ. The quotient is truncated towards zero.

The number of decimal places in the result, stored in A scaling, is the original A scaling minus the number of decimal places in the divisor (q).

If the magnitude of the remainder is greater than or equal to half the magnitude of the divisor then the rounding flag is set to 1, if the divisor and dividend have the same sign, and to -1 if different. Otherwise it is cleared.

Overflow will occur in the following circumstances:

- The magnitude of the divisor is greater than or equal to $2^{31} \times 10^{-q}$;
- The magnitude of the dividend is greater than or equal to the magnitude of the divisor times ($2^{31} \times 10^{-r}$) where r is the number of decimal places in the result, as defined above.

These conditions for overflow are equivalent to saying that both the divisor and the quotient must be capable of being stored in a four-byte numeric field, and that neither may have the largest negative value which can be held in that field.

If the store option is specified in the qualifier then the accumulator is scaled to match the target field and stored in the target. The rounding flag is cleared. Overflow will occur if the result is too large for the target field (see Table 4.2.1.1).

4.5.7.1 Exceptions

The following exceptions may occur:

7 Illegal Qualifier
11 Overflow
12 Memory Violation
16 No Base

4.5.7.2 Registers and Storage Modified

The following registers and storage are modified:

A accumulator
Target field (if store option specified)

I, L and Q are cleared

4.5.7.3 Programming Notes
Division by zero gives an overflow exception. Note that the scaling of the result may be negative.

4.5.8 Store Unsigned
Store Unsigned stores the accumulator in the target field without checking for overflow. No scaling is allowed: if it is necessary a scaling exception may occur.

4.5.8.1 Exceptions
The following exceptions may occur:

7 Illegal Qualifier
12 Memory Violation
16 No Base
17 Scaling

4.5.8.2 Registers and Storage Modified
The following registers and storage are modified:

Target location in store
I, L and Q are cleared

4.5.8.3 Programming Notes
The Store Unsigned instruction is used to perform unsigned arithmetic, in particular pointer addition or subtraction.

4.5.9 And - AND Accumulator
And performs a logical AND of the target field with the accumulator. If the target field is shorter than 8 bytes, it is considered to be extended on the left with zero bytes. No scaling is allowed: if any is necessary a scaling exception may occur.

If the store in target flag is set the accumulator is stored in the target field without checking for overflow (i.e. a Store Unsigned is performed).

4.5.9.1 Exceptions
The following exceptions may occur:

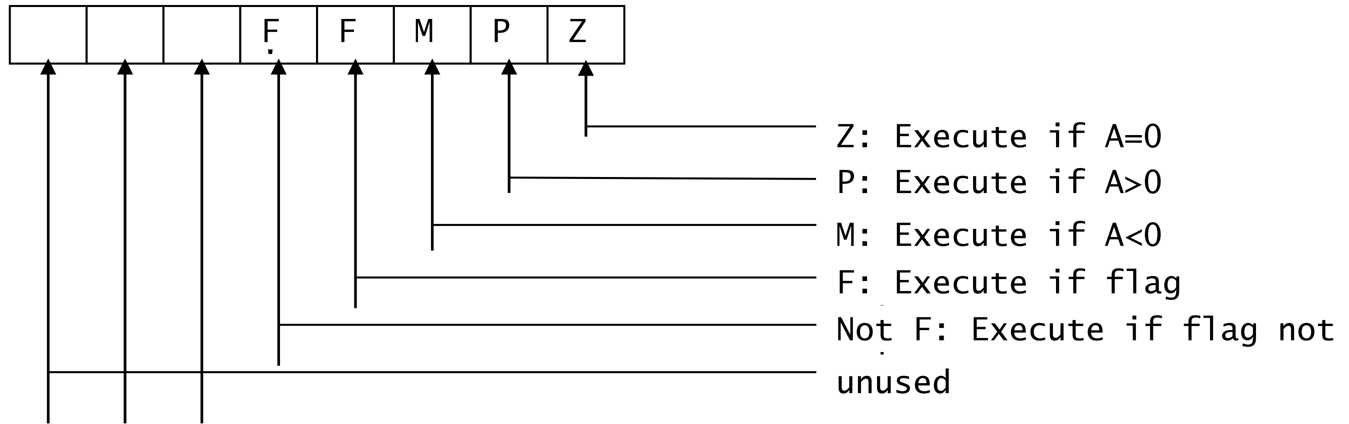
7 Illegal Qualifier
12 Memory Violation
16 No Base
17 Scaling

4.5.9.2 Registers and Storage Modified
The following registers and storage are modified:

Accumulator
Target field (if store option specified)
I, L and Q are cleared

4.6 Transfer of Control Instructions
The transfer of control instructions are used for branching (Jump instruction); subroutine entry and exit (Call and Exit); and for returning control to the monitor (Stop).

If, when the instruction is executed other than immediately following a Resume, the interrupt-pending flag is non-zero, and the flag register is zero, and the privilege status register is -1, then the exception number given by the interrupt-pending flag is returned and the interrupt-pending flag is reset to zero.



Permissible Combinations

A Not zero is P+M = #06
A Not negative is Z+P = #03
A Not positive is Z+M = #05
Unconditional is Z+P+M = #07
Unconditional, clearing F, is F+not-F = #18

Figure 4.6 - Transfer of control instruction qualifier

The instruction qualifier is used to determine whether the instruction succeeds, or whether the next sequential instruction is to be taken. Figure 4.6 shows the format of the instruction qualifier. A condition bit is set to 1 to specify that its associated condition is to be applied, zero otherwise. Thus:

- With not-F set and other condition bits zero, transfer of control only takes place if the flag register is zero;
- With the zero, positive and negative condition bits set (qualifier=7) the transfer of control will always take place, i.e. it is an unconditional instruction;
- With all condition bits set to zero, transfer of control can never take place. The instruction is an effective no-operation.

Figure 4.6 also shows the permissible combinations of conditions; the effect of other combinations is not defined.

The transfer of control instructions are valid when a tentative exception is pending providing they are conditional on F or not-F. Such an instruction, which tests F, subsequently turns it off and thereby clears the pending tentative exception.

When a transfer of control is to take place the address of the instruction is saved so that it can be placed in the diagnostic logout area if an immediate exception occurs.

4.6.1 Jump - Jump Conditionally

If the Jump instruction does not succeed the next sequential instruction is executed.

If the Jump succeeds then control is transferred to the address given by the two-byte pointer at the target address. If this address is odd an illegal branch exception will be returned.

4.6.1.1 Exceptions

The following exceptions may occur:

1 Break
8 Illegal Branch
15 Advance
16 No Base
20-29 Interrupts

4.6.1.2 Registers and Storage Modified

The following registers and storage are modified:

I, L and Q are cleared
If the instruction is conditional on the F register then the F register is cleared
The interrupt-pending flag may be cleared

4.6.2 Call - Call Subroutine

If the Call does not succeed the next sequential instruction is executed.

If the Call succeeds a new link stack entry is initiated, an immediate link stack exception taking place if all 28 entries are already in use. If the two-byte target, a pointer, is odd an illegal branch exception occurs. Otherwise the pointer to the next sequential instruction following the Call is remembered in the first two bytes of the stack entry, the pointer to the target of the transfer instruction in the next two bytes, and control is passed to the instruction addressed by the target pointer.

4.6.2.1 Exceptions

The following exceptions may occur:

1 Break
8 Illegal Branch
14 Link Stack
15 Advance
16 No Base
20-29 Interrupts

4.6.2.2 Registers and Storage Modified

The following registers and storage are modified:

Link Stack and Link Stack Index (if Call succeeds)
I, L and Q are cleared
F is cleared if the instruction is conditional on the F register
The interrupt-pending flag may be cleared

4.6.2.3 Programming Note

Programs are normally entered by means of a Call from the monitor. Hence there are 27 levels of subroutine available to the application program. However, 12 of these levels are reserved for use by the

monitor, for example when called to perform I/O, leaving 15 levels available to the application program.

4.6.3 Stop - Return to Monitor

If the Stop does not succeed then the next sequential instruction is executed.

If Stop succeeds its two-byte target is established in the completion code field in the system area, and an immediate program exception (Stop, number 5) takes place.

4.6.3.1 Exceptions

The following exceptions may occur:

1 Break
5 Stop
15 Advance
16 No Base
20-29 Interrupts

4.6.3.2 Registers and Storage Modified

The following registers and storage are modified:

Completion Code in the system area
I, L and Q are cleared
F is cleared if the instruction is conditional on the F register
The interrupt-pending flag may be cleared

4.6.3.3 Programming Notes

Global Cobol is responsible for completion code assignment. The senior byte of the completion code is generally used to identify the component responsible for detecting the stop condition. The junior byte further particularises the reason for the stop when a single component is sensitive to a number of different stop conditions.

4.6.4 Exit - Return from Subroutine

If the Exit does not succeed the next sequential instruction is executed.

If Exit succeeds its two-byte target is established in the completion code field in the system area, and if the code thus established is non-zero a tentative exception (Exit, number 9) takes place causing the F register to be set on. If the privilege status register is equal to the link stack index the privilege status is set to -1. The top entry of the link stack is popped, and the instruction pointed to by the entry (the next sequential instruction of the previous Call) is interpreted. (In the event that the link stack is empty, control is returned to a machine specific error routine with the completion code in the parameter register, an Exit from the highest level routine being the conventional means of ending interpretation.)

If Exit returns a non-zero completion code and the caller contains logic which tests the F register, all will be well. Otherwise, since the caller is unaware of the non-zero completion code the tentative exception will become actual.

4.6.4.1 Exceptions

The following exceptions may occur:

1 Break
9 Exit
15 Advance
16 No Base
20-29 Interrupts

4.6.4.2 Registers and Storage Modified

The following registers and storage are modified:

Privileged Status register (only if Exit instruction successes)
Link Stack Index (only if Exit instruction successes)
Completion Code in system area F (only if Exit instruction successes)
I, L and Q are cleared
F will be cleared if the instruction is conditional on the F register and does not reset it
The interrupt-pending flag may be cleared

4.6.4.3 Programming Notes

Since programs are entered via Call from the monitor program, the top link on the stack will be a return to the monitor, hence an Exit with the link stack empty can normally only be issued by the monitor.

Global Cobol is responsible for completion code assignment. The senior byte of the completion code identifies the component responsible for detecting the Global Cobol 'exception'. The junior byte of the code becomes the PIC 9(2) field $$COND, the Global Cobol 'exception condition'.

4.7 Status Switching Instructions

The status switching instructions are special purpose instructions which cannot be indexed or qualified by the Q register.

Resume allows the user program to be restarted at the failing instruction following an exception. Escape enables machine code programs to be executed from Global Cobol. Trace is a no-operation whose purpose is to allow a five-character section or entry point identifier to be optionally embedded in Global Cobol code.

4.7.1 Resume - Resume Failed Program

The Resume instruction is provided for use by debug routines written in Global Cobol. Its qualifier is unused and it is set up as a two-byte immediate instruction. It cannot be indexed. The I and Q registers are ignored. When it is interpreted the status preserved in the diagnostic logout area (and possibly modified by the debugging procedure) is restored.

Interpretation proceeds by retrying the failing instruction pointed to from the diagnostic logout area. A trap exception or interrupt on the first instruction interpreted following a Resume is suppressed in order to allow interpretation to continue after a trap exception without having to clear the trap, and to allow at least one instruction to be executed after an Advance interrupt.

4.7.1.1 Exceptions

None.

4.7.1.2 Registers and Storage Modified

The following registers and storage are modified:

A register, Privilege Status register
I, L, Q, F, T length, T pointer
Completion code and exception codes in the system area
Link Stack and Link Stack Index
Parameter Stack and Parameter Stack Index

4.7.1.3 Programming Notes

If the instruction to be resumed used the I, L or Q registers these may have to be reset before resuming as they might have been cleared before the exception was detected. This is not necessary, however, following a trap exception.

The flag register in the logout area will always be zero and must be reset if required.

The Global Cobol user is advised to resume his program on a statement boundary following any exception apart from trap or break. Global Cobol, therefore, does not currently support modification of the I, L, Q or F registers in the diagnostic logout area.

4.7.2 Escape - Escape into Machine Code

The Escape instruction is a two-byte immediate mode instruction whose qualifier is unused. It must be immediately followed by the machine code program which is to be executed. Escape is coded at the beginning or near the beginning of a routine which is entered from Global Cobol via a Call.

When the Escape instruction is interpreted a machine code interface area is set up. This involves converting any pointers in the parameter stack to machine addresses in this area. The machine code program is then entered with the parameter address register containing the machine address of the interface area. The parameter stack is always cleared by an Escape instruction.

The program can obtain parameters and communicate results using the areas whose addresses were obtained from the parameter stack.

The machine code program completes by causing the interpreter to execute a pseudo-Exit or pseudo-Stop instruction by passing control to either the exit address or stop address defined in the interface area. The target of these pseudo-instructions is the two-byte completion code at the start of the interface area. This code will be zero on entry to the machine code program.

If the routine completes by passing control to the exit address then the completion code in the interface area is moved to the completion code field in the system area, and if the code thus established is non-zero a tentative exception (Exit, number 9) takes place causing the F register to be set on. The top entry of the link stack is popped, and the instruction pointed to by the entry (the next sequential instruction of the previous Call) is interpreted.

If the routine completes by passing control to the stop address then the completion code in the interface area is moved to the completion code in the system area and an immediate program exception (Stop, number 5) takes place.

The instruction qualifier should be zero if the assembler routine to be executed resides within the Global Cobol address space. A non-zero value is used in some interpreters to indicate routines in the Global nucleus which are outside the normal address space (this affects details of the interface to the routine, such as how segment registers are established).

4.7.2.1 Exceptions
The following exceptions may occur:

5	Stop
9	Exit

4.7.2.2 Registers and Storage Modified
The following registers and storage are modified:

Machine Code Interface area
Parameter Stack Index
Completion Code
Link Stack Index (pseudo-Exit only)

4.7.2.3 Programming Notes
Details of the interface to the machine code routines will depend on the microprocessor being used.

4.7.3 Trace - Trace Identifier
The Trace instruction enables trace information to be embedded within the code of a routine. The Trace instruction is always six bytes long and consists of a 1-byte operation code followed by five bytes which are ignored. These five bytes will normally be used to hold a 5-character ASCII identifier for use by the debugging system.

A Trace instruction is always a 'no-operation' and the interpreter continues by executing the next sequential instruction.

4.7.3.1 Exceptions
None.

4.7.3.2 Registers and Storage Modified
No registers or storage are modified.

4.7.3.3 Programming Notes
A Trace instruction containing the section name will normally be generated by Global Cobol following each SECTION or ENTRY statement. The second pointer of each link stack entry can be used to access these instructions so that the debugging system can print out the names of the routines which called the current routine.

4.8 Parameter Stack Instructions
The parameter stack instructions, Push, Pop and Pop List, maintain the parameter stack used for passing parameters by name and in the implementation of based variables. The parameter stack is also affected by the Escape instruction.

The use of the qualifier is different for each of the instructions, and is given in the instruction descriptions.

4.8.1 Push - Push Parameter onto Stack

The Push instruction places a pointer to its target on the top of the parameter stack. If the address mode is immediate or the instruction is indexed the qualifier must be the length of the target in bytes. The qualifier is not used for the other address modes.

If the parameter stack is full (7 items already on stack) then a parameter stack exception is returned.

4.8.1.1 Exceptions
The following exceptions may occur:

13 Parameter Stack

4.8.1.2 Registers and Storage Modified
The following registers and storage are modified:

Parameter Stack
Parameter Stack Index
I, L and Q are cleared

4.8.1.3 Programming Notes
Note that if the Push instruction is used with a fixed point immediate operand, the length in bytes must be supplied as the qualifier rather than the normal qualifier for fixed point numbers.

4.8.2 Pop - Pop Parameter from Stack
The Pop instruction removes the top pointer from the parameter stack and places it in its two-byte target. The qualifier is not used.

If the parameter stack is empty a parameter stack exception occurs.

4.8.2.1 Exceptions
The following exceptions may occur:

13 Parameter Stack
16 No Base

4.8.2.2 Registers and Storage Modified
The following registers and storage are modified:

Parameter Stack Index
2-byte target field
I, L and Q are cleared

4.8.3 Pop List - Pop All Parameters from Stack
Pop List is an instruction provided to simplify and optimise the processing of the USING clause when it appears in a Global Cobol ENTRY statement. It also provides the mechanism by which a routine can be made privileged.

The junior four bits of the qualifier form a number, n. The senior bit (of the junior byte) is the privileged flag. The remaining 3 bits are currently unused.

If the privilege status register is -1 (unprivileged) and the privileged flag in the qualifier is set the current value of the link stack index is stored in the privilege status register.

If the privilege status register is not -1 (privileged) and the privileged flag in the qualifier is zero the privilege status register is reset to -1.

If n is not equal to the parameter stack index then a parameter stack exception occurs and the parameter stack is not altered.

If n is zero then no further processing occurs, i.e. the instruction simply checks that the parameter stack is empty. Otherwise n is in the range 1 to 7 and the target is a list of n pointers. The instruction pops the top entry from the stack and stores it at the location addressed by the n'th pointer; the next entry is stored at the location addressed by pointer n-1; and so on.

4.8.3.1 Exceptions

The following exceptions may occur:

13   Parameter Stack

4.8.3.2 Registers and Storage Modified

The following registers and storage are modified:

Storage locations addressed by the list of pointers
Parameter Stack Index
Privilege Status register
I, L and Q are cleared

4.9 Register Load Instructions

The three register load instructions enable the user program to set up the I, L and Q registers. The qualifier in these instructions has the same format as that used in an Arithmetic Load. The value placed in the register is the integer part of the target. An illegal load immediate program exception takes place if this is not in the range 1 to 32,767 inclusive, or 0 to 65,535 for the Q register.

If the qualifier is not a valid arithmetic instruction qualifier (see chapter 5) then an illegal qualifier exception will occur.

4.9.1 Load Index - Load I Register

Load Index moves the integer part of the target into the I register, providing it is in the valid range 1 to 32,767.

The non-zero I register will be used in indexing the subject instruction immediately following Load Index (as described in 4.2.2.1) unless it is Load Qualifier, Load Length, Trace, Escape or Resume. In the case of Load Qualifier, Escape and Trace, I remains undisturbed and available to index the next instruction eligible as subject. Resume restores the I register from the diagnostic logout area, ignoring its initial contents.

Once the I register is used for indexing it is zeroised. This prevents further indexing taking place until a subsequent Load Index instruction has re-established the register.

A Load Index instruction can be the subject of a preceding Load Index instruction. In this case the first instruction simply sets the index used in establishing the target of the second, the integer part of which becomes the new index.

4.9.1.1 Exceptions
The following exceptions may occur:

3	Illegal Load
7	Illegal Qualifier
16	No Base

4.9.1.2 Registers and Storage Modified
The following registers and storage are modified:

I register
L and Q registers are cleared

4.9.1.3 Programming Notes
If both the I and Q registers are to be set up for an instruction, the Load Index must precede the Load Qualifier instruction.

If both the I and L registers are to be set up for an instruction, the Load Index must precede the Load Length.

If all three I, L and Q registers are to be set up for an instruction, the instructions must be in the order Load Index, Load Length, Load Qualifier. If necessary, the Load Length instruction could be preceded by a Load Qualifier instruction to establish the qualifier of the length field.

4.9.2 Load Qualifier - Load Q Register
Load Qualifier is used to establish a qualifier value in the Q register which will be used when the next instruction executes in place of the value normally obtained from its second byte. The senior 8 bits of the Q register are ignored except in byte string instructions where Q can be employed to specify a string length of up to 65,535 bytes.

Once the Q register has been used for qualification it is zeroised. This prevents its further use until a subsequent Load Qualifier instruction re-establishes it. Note however that Escape and Trace do not use the Q register, and therefore do not clear it.

The index register is ignored by Load Qualifier and therefore one or more Load Qualifier instructions may be interposed between a Load Index and the instruction which is the subject of the indexing.

4.9.2.1 Exceptions
The following exceptions may occur:

7	Illegal Qualifier
16	No Base

4.9.2.2 Registers and Storage Modified
The following registers and storage are modified:

Q register

4.9.2.3 Programming Notes
A Load Qualifier instruction is normally used either to set up a qualifier greater than 255 or when the qualifier to be used is a variable.

Note that if the Q register is non-zero when a Load Qualifier instruction is executed that instruction itself uses the value in the Q register for its qualifier.

In V5.2 interpreters no check is made on the value of the target. In particular, it is possible to load a zero qualifier.

The effect of using a Load Qualifier instruction in front of a short immediate mode instruction is not defined.

4.9.3 Load Length - Load L Register

Load Length moves the integral part of the target into the L register, provided it is in the range 1 to 32,767. This length will be used when the next instruction is executed in place of the operand length by which the index is normally multiplied.

Once the L register has been used it is zeroed. This prevents its further use until a subsequent Load Length instruction re-establishes it.

The index register is ignored by a Load Length instruction so that a Load Length instruction can be placed between a Load Index instruction and the instruction which is to be indexed.

4.9.3.1 Exceptions

The following exceptions may occur:

3 Illegal Load
7 Illegal Qualifier
16 No Base

4.9.3.2 Registers and Storage Modified

The following registers and storage are modified:

L register
Q register is cleared

4.10 Special Purpose Instruction

4.10.1 Call Display - Call Display SVC

This is a special instruction which can only be used to call the character display routine, SVC 5 (SVC = **S**uper**v**isor **C**all). To invoke SVC 5 using normal intermediate code instructions requires 3 instructions: a Set, a Push and a Call. Hence this instruction saves 8 bytes for each DISPLAY statement for a character variable.

Call Display establishes the address of the target in the T pointer register. The length of the target is given by the low-order 7 bits of the qualifier (or Q register if set), and this length is stored in the T length register. The target may be indexed or a literal.

A pointer to the start of the instruction is placed on the top of the parameter stack. If the parameter stack is full then a parameter stack exception is returned.

The I, L and Q registers are cleared and execution continues as if an unconditional Call instruction (qualifier = 7) had been executed with address mode 0 and the target a pointer to location -12 (hex FFF4).

However, the interrupt-pending flag is not tested and so a break cannot occur.

4.10.1.1 Exceptions

The following exceptions may occur:

13 Parameter Stack
14 Link Stack
16 No Base

4.10.1.2 Registers and Storage Modified

The following registers and storage are modified:

T pointer
T length
Parameter Stack
Parameter Stack Index
Link Stack
Link Stack Index
I, L and Q are cleared

4.10.1.3 Programming Note

The qualifier has the standard console control byte format, with the length in the low order 7 bits, and the top bit set to 1 for a display on a new line, or zero for a display on the same line.

Note that if this instruction is immediately preceded by a Load Qualifier instruction, then the length of the target is given by the Q register, but the number of characters to be displayed is given by the instruction qualifier.

5. Metajob Management

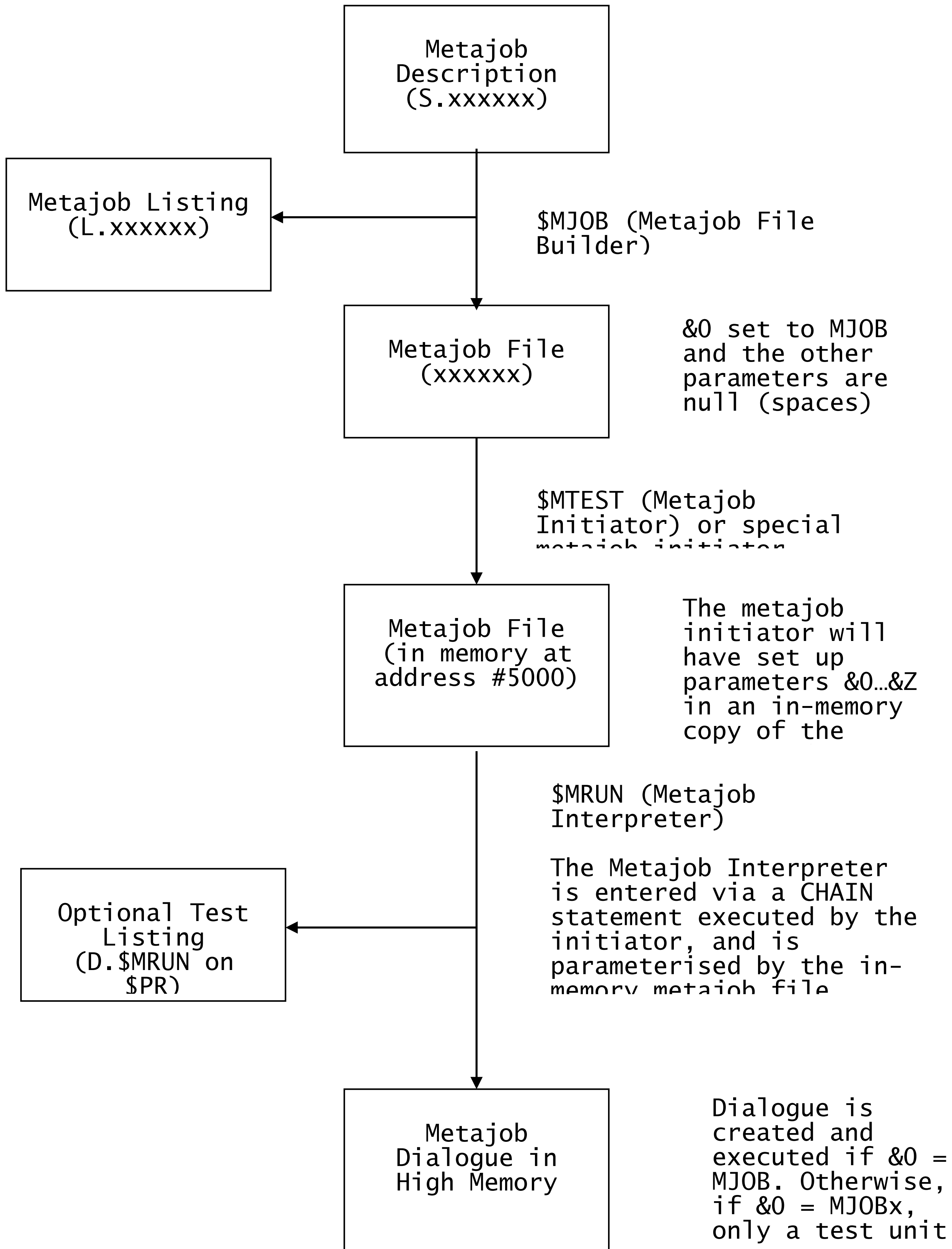


Figure 5.1.1 - The Metajob management system

5.1 Foreword

The Metajob Management System allows you to write programs in a high-level metajob language which then generates job management dialogue according to the values set up in parameter fields. It is particularly designed for the production of dialogue to copy or install Global software for a variety of configurations, and contains many special statements for running the file utility, $F, and the librarian, $LIB. However, it is not restricted to these utilities, and can be used to generate any dialogue required.

A metajob description consists of directives and statements. Standard structured programming IF...ELSE...END constructs are provided to allow statements to be executed conditionally, according to parameter settings. Furthermore, most statements use operands which can themselves be parameterised and which are eventually evaluated when the metajob is run. The parameter values are established by a user-provided Global Cobol program known as a metajob initiator, which can either prompt the operator for data or obtain values from environmental information such as system variables. Great flexibility is achieved by the coupling of this generalised metajob system with a specialised initiator dedicated to a particular purpose, such as Global System Manager installation.

5.1.1 The Metajob Management System

The three command programs $MJOB, $MTEST and $MRUN, illustrated in Figure 5.1.1, constitute the Metajob Management System.

$MJOB, the metajob file builder, inputs a source file containing the metajob program - known as the metajob description - and 'compiles' its statements into a metajob file, which is a conventional program file containing only a data division. At the same time a metajob listing is produced on a specified unit. The listing contains standard headings, the original source lines with sequence numbers, and error and warning messages if any of the lines are faulty or suspect.

(In the figure, in order to save space, we have used the abbreviation 'mjob' in place of 'metajob'. This abbreviation is also used in dialogue, error and warning messages, and in some of the operating descriptions which follow.)

The second program of the system, $MTEST, is a general-purpose metajob initiator, which allows you to set any parameter from the terminal. It loads a specified metajob file into high memory (locations #5000 onwards) and establishes parameter values in a table area reserved for this purpose at the very beginning of that file, preceding the compiled statements. When all the values are set up, $MTEST chains to $MRUN, the metajob interpreter.

$MTEST is useful for checking out a metajob during testing, but it is not suitable for production work. You will normally replace it with your own dedicated initiator which will load the metajob file, establish the parameters and chain to $MRUN in the same way. Alternatively you can run $MTEST under job management to provide a specialised initiator.

$MRUN processes the metajob file passed to it in high memory. It fetches the compiled statements one by one, evaluating any parameterised operands using the values established by the initiator. Like any true program, the metajob may contain conditional jump

instructions (derived from IF statements) which are effective, or not, according to parameter values.

When used in 'live' mode $MRUN creates dialogue using the JOB$ system routine, and the dialogue is actioned as soon as interpretation is complete. However, by setting a reserved parameter, &0, to a special value, you may instead output a test listing to the standard printer, $PR, rather than produce dialogue for JOB$. The listing contains the statements processed, and shows their operands with the parameters evaluated. The sequence number of each statement is output so that you can correlate information from the test listing with that from the metajob listing produced earlier by $MJOB. Optionally, you can have the dialogue each statement will generate in live mode printed on the test listing.

To simplify the implementation the system performs only limited validation of your metajob. $MJOB itself checks the integrity of the structured programming constructs and ensures that statements are recognisable and have the correct number of operands. $MRUN performs a number of sequencing checks on the information it is processing and will display (and optionally print) an error message and terminate if one of these is invalidated. However, none of the testing takes account of the detailed dialogue involved. For example, if an operand ought to be a three digit unit address, but in fact contains a five character string, all will be well as far as $MJOB and $MRUN are concerned. The error will either become apparent when you examine the test listing or will be found when the faulty dialogue is eventually actioned in live mode.

5.1.2 Structure of this Manual

Immediately following this Foreword, section 5.2 explains how you write a metajob description, and describes the various directives and statements available. The statements are subdivided into related groups and each group is treated in a section of its own. You should read the whole of the chapter, referring to the example metajob in Appendix A. Later, when you come to code a metajob Table 5.2.2 gives you quick access to the specification of individual statements.

Section 5.3 explains how to use $MJOB to produce a metajob file from a metajob description, and section 5.4 describes how you can use $MTEST to set up parameters and run the metajob file you have created.

Section 5.5 describes how to write your own metajob initiator to replace $MTEST in a live environment.

Appendix A contains a complete worked example which is referred to throughout this manual. The source and metajob listing are provided, together with test output created both by an example initiator and a run of $MTEST. The example is not intended to be completely realistic, but it contains fragments which are likely to be used in Global distribution and installation, and shows most of the possible combinations of statements and operands, together with the resulting dialogues. There is also a compilation listing of the example initiator, which should be studied in conjunction with section 5.5.

Appendices B and C contain explanations of the error and warning messages that may be produced by $MJOB and $MRUN respectively.

5.2 The Metajob Description

A metajob description is a text file containing lines of up to 72 characters. Each line is in free format, with leading blanks and tabs before the first significant character being ignored. Within the body of a line, a blank, tab or any combination of blanks and tabs may serve as a separator between different elements.

Lines themselves are of three types: directives, comments or statements. The first line of a metajob description must be an MJOB directive, and the last line an ENDMJOB directive. Between these there may be any combination of comment lines, statement lines, or PAGE directive lines.

There is an example metajob description listed in Appendix A1, and you may find it useful to briefly examine it at this stage and refer to it as you proceed through this chapter.

5.2.1 Comments and Directives

5.2.1.1 Comment Lines

Lines whose first significant character is an asterisk may be coded anywhere between the MJOB and ENDMJOB directive. They are listed but otherwise ignored:

```
* THIS IS A COMMENT
```

5.2.1.2 Comments on Instructions

Comments may appear to the right of the last operand of a statement (or of its name, if there are no operands). They can be introduced by an asterisk separated from the last operand or name by one or more blanks or tabs:

```
PERFORM SYSDEV                     * CREATE SYSDEV VOLUME
OUTPUT SYSOPT 101                  * SELECT SYSOPT
```

Comments may **not** be coded on the directive lines, MJOB, PAGE andENDMJOB.

5.2.1.3 The MJOB Directive

The MJOB directive must be coded as the very first line of the metajob description. Its format is:

```
MJOB mjob-id title
```

The mjob-id is really only present to enable you to document the name that you intend to give to the metajob file when it is produced by the $MJOB command from this metajob description. The name of the file is actually determined from information keyed to the $MJOB command, but you will normally choose to make this the same as the mjob-id. An mjob-id is usually six characters or less, and the metajob description source file should be named S.mjob-id, e.g. S.EXMJOB in the case of the example in Appendix A1. The mjob-id is terminated by the first blank or tab following it.

The title begins with the first character after the mjob-id which is not a blank or tab. It can be up to 30 characters in length and may contain embedded blanks. If more than 30 characters are present only the first 30 will be used and if the end of the line is met before 30 characters have been encountered the title will be made up to 30 characters with trailing blanks. The title is stored in the program

header of the metajob file, so that it can become a member title if this file is later incorporated into a program library.

The title also appears in the heading printed on the metajob description listing, unless it is overridden by the title from a subsequent PAGE directive.

5.2.1.4 The PAGE Directive

You can use the PAGE directive to cause $MJOB to advance the metajob listing to a new page and print a header. You code:

```
PAGE ["character-string"]
```

The optional character-string may be supplied to change the title of the listing (which is initially taken from the MJOB directive). If a title is specified it will appear in all subsequent page headings until a PAGE statement is met which contains a new title.

5.2.1.5 The ENDJOB Directive

The ENDJOB directive must be coded at the very end of the metajob description. Normally it should be preceded by an EXIT or JUMP statement to transfer control to some other point within the metajob. However, if such a statement is not present $MJOB will automatically implant an EXIT statement to prevent the interpreter accessing information beyond the end of the file.

Name	Operands	Reference	Size
IF	string-1 [string-2]	5.2.3.1	24/32
ELSE		5.2.3.1	32
END		5.2.3.1	16
SECTION	name	5.2.3.2	16
PERFORM	name	5.2.3.3	16
EXIT		5.2.3.3	8
JUMP	name	5.2.3.4	16
SUPPRESS		5.2.4.1	8
VERIFY	[OFF]	5.2.4.2	8/16
INPUT	[volume-id] unit-id	5.2.4.3	16/24
OUTPUT	[volume-id] unit-id	5.2.4.3	16/24
COMMAND	[OFF]	5.2.4.4	8/16
SERIAL	[number] [expiry date]	5.2.5.1	8/16/24
PROTECT	file-id	5.2.5.2	16
COPY	[file-id] [file-id|size|<GROUP>]	5.2.5.3	8/16/24
LOAD		5.2.5.4	8
INSTALL	file-id unit-address	5.2.5.5	24
PATCH	file-id SYSRES-address stack/bank	5.2.5.6	32
ALLOCATE	file-id size	5.2.5.7	24
MOUNT		5.2.5.8	8
CHECK		5.2.5.9	8
DELETE	file-id|selection	5.2.5.10	16
INIT	[volume-id]	5.2.5.11	8/16
LIBRARY	file-id [size]	5.2.6.1	16/24

OLDLIB	OLDLIB | file-id [size]	5.2.6.2	16/24
MERGE	file-id [name|<volume-id>]	5.2.6.3	16/24
TAG	Identifier	5.2.6.4	16
INVOL	volume-id [unit-id]	5.2.6.5	16
ENDLIB		5.2.6.6	8
EXTRACT	library-id member	5.2.6.7	24
RUN	[program-id]	5.2.7.1	8/16
RESPOND	response (up to 64 chars)	5.2.7.2	8+length
MESSAGE	message (up to 64 chars)	5.2.7.3	8+length
PAUSE	message (up to 64 chars)	5.2.7.4	8+length
SET	parameter [value]	5.2.8.1	16/24
MASK	parameter mask	5.2.8.2	16/24

Table 5.2.2 Metajob Description Statements

5.2.2 Statements, Parameters and Operands

Table 5.2.2 lists the available metajob statements, showing the names and the values of the zero, one, two or three operands that the statement takes. The name must be the first significant information on the statement line, and it must be separated from the operands (if any) by one or more blanks and tabs. The name or last operand may optionally be followed by one or more blanks and tabs and then an asterisk introducing a comment.

The table shows the statements divided into groups, each of which is described in a separate section below. This section itself explains the general conventions applying to parameters and operands which affect every type of statement.

The column in Table 5.2.2 indicates the size of each statement; this varies in multiples of 8 as operands are included or omitted.

5.2.2.1 User Parameters

There are 35 user parameters known as &1,...&9,&A...,&Z. That is, the parameter name is made up from the &-character concatenated with either a digit other than 0, or an upper case letter. Each parameter references a unique 8-byte field whose value can be determined by your metajob initiator. Parameters can be coded in the operands of statements so that they vary in accordance with information set up by the initiator.

Although a parameter's value is stored in a 8-byte field, its effective length may be between 0 and 8 bytes, since an ASCII blank is considered to terminate the parameter. A parameter whose value does not contain a blank is considered to be the full 8 bytes in length. On the other hand, one whose value starts with a blank is 0 bytes long. Such a parameter is said to be **null** (or undefined).

If a parameter value needs to contain one or more embedded blanks, these must be represented by @ characters, which are converted to spaces by job management. An @ character itself cannot be supplied as dialogue.

5.2.2.2 The Mode Parameter, &0

The mode parameter, &0, is similar to a user parameter in as much as it can assume an 8-byte value. However it is currently constrained to

one of three valid values: MJOB; MJOB1; or MJOB2. These are used to determine whether the metajob runs live or in one of two test modes.

When the mjob file is built &0 is set to MJOB, and if this value is unchanged the metajob operates in live mode, using the JOB$ system routine to generate dialogue which is eventually actioned.

The initiator may change the mode parameter to either MJOB1 or MJOB2 to cause the metajob to run in a test mode in which JOB$ is not used. Instead, the metajob statements to be executed are listed on file L.$MRUN on $PR, the standard printer.

The difference between MJOB1 and MJOB2 modes is that in the first case only the metajob statements themselves are listed, while in the second case the dialogue which the statements would generate is printed as well. The examples in appendices A4 and A5 are listings output in MJOB2 mode.

Note that if you attempt to execute a metajob and &0 does not contain one of the permissible values MJOB, MJOB1 or MJOB2 your job will terminate in error.

5.2.2.3 Operands

A statement may contain zero, one, two or three operands. An operand is made up of a combination of non-blank ASCII graphic characters and parameters. It is separated from the statement name, another operand or a comment by one or more blanks or tabs. A maximum of 8 characters or parameters may be specified as an operand: if more characters or parameters are present they will simply be ignored. For example, the following are all maximum size operands:

```
&A&B&C&D&E&F&G&H

&A&BCDEF&G&H

ABCDEFGH
```

When a parameter is coded, its actual value will be substituted when the operand is evaluated by the metajob interpreter, $MRUN. A null parameter, zero bytes in length, will effectively be ignored. In addition, only the first 8 characters of the operand resulting from parameter substitution will be used. Thus, suppose, for the operands coded above, the parameters were defined as follows:

```
&A = 12
&B = 3
&H = 456
others = null
```

Then the first operand would evaluate as 123456, the second as 123CDEF4, and the third, which contains no parameters, would remain as ABCDEFGH.

An omitted operand, or one which is made up entirely of null parameters, is known as a **null** operand.

The operand of a MESSAGE or RESPOND statement is treated differently. It can be up to 64 characters long and can contain spaces. It is terminated by the end of the line, or by the start of a comment, trailing spaces being ignored.

5.2.2.4 Operand Restrictions and &*

Because of the way the & character is used to identify parameters, this character cannot be used in isolation within an operand.

You should note, too, that an operand cannot begin with an asterisk, since if it does, it will be taken to be a comment. However, you may code the special sequence &* in place of the asterisk character. This has the value * when the operand is evaluated.

Thus, for example, if you need to run the command $CLIENT from the unit assigned to $P you should code the statement:

```
RUN &*CLIENT
```

The operand evaluates to *CLIENT and the statement (explained later in 5.2.7.1) generates the dialogue:

```
:<ESCAPE>
GSM READY:*CLIENT
```

to run $CLIENT from $P.

5.2.3 Structural Programming Statements

The statements described in this section do not (usually) cause dialogue to be generated but instead allow you to write a metajob using conventional conditional structures and performed subroutines.

5.2.3.1 Conditional Structures

Conditionals are created using the IF, ELSE and END statements using the following familiar format:

```
IF condition
.
.    (statements to be executed if the condition is true)
.
[ELSE
.
.    (statements to be executed if the condition is false)
.     ]
END
```

Conditional structures may be nested, for example:

```
IF condition 1
     IF condition 2
.
.         (statements executed if condition 1 and condition 2 are
both true)
.
     END
ELSE
     IF condition 3
.
.         (statements executed if condition 1 is false and
condition 3 true)
.
     ELSE
.
```

```
.        (statements executed if condition 1 is false and
condition 3 is false)
.
     END
END
```

The maximum depth of nesting is 32 levels.

The condition may consist of either one or two operands. If the first operand is null, the condition is false. If the second operand is null or omitted, and the first is not null, then the condition is true. Otherwise, the condition is true if and only if the character string represented by the second operand is contained within the character string corresponding to the first operand. In particular, if the two operands are the same length (and not null) the condition is true if they are equal, and false otherwise.

In the following example, &M is null if a single user system is being installed, or MU for a multi-user system. The effect is to copy the appropriate monitor, $MONITOR or $MONITMU, from the input volume to the output volume. (The COPY statement itself is explained later: the example is to illustrate a conditional structure used in conjunction with parameter evaluation.)

```
IF &M
     COPY $MONIT&M
ELSE
     COPY $MONITOR
END
```

The test could also have been coded:

```
IF &M MU
```

although this would generate 8 bytes more metajob.

This second, simple example sets verification on, using the VERIFY statement, if the parameter &V, which contains a single digit, is 1, 2 or 4:

```
IF 124 &V
     VERIFY
END
```

The final example writes the message ONE ON, BOTH ON or NONE ON, depending on whether one, both or neither of &A and &B are non-null:

```
IF &A
     IF &B
          MESSAGE BOTH ON
     ELSE
          MESSAGE ONE ON
     END
ELSE
     IF &B
          MESSAGE ONE ON
     ELSE
          MESSAGE NONE ON
     END
END
```

5.2.3.2 Sections and Paragraph Names

A metajob can be subdivided into any number of self-contained, individually named sections using the SECTION statement. This takes one compulsory operand, the section name. You simply code:

```
SECTION name
```

You can insert paragraph names by coding name., thus:

```
name.
```

Like any other operand, the name can be up to 8 characters in length. However, **unlike** any other type of operand the name used in a SECTION statement or paragraph name should not contain parameters, since it cannot vary dynamically. Each name must be unique, not previously used in a SECTION statement or paragraph name.

A section is considered to be delimited by another SECTION statement or the ENDMJOB directive. Conditional structures should not span sections.

5.2.3.3 Subroutining using PERFORM and EXIT

The PERFORM and EXIT statements allow you to execute the statements introduced by a SECTION statement as a subroutine. You code:

```
PERFORM name
```

to pass control to the section of the same name. The name operand following the PERFORM statement may contain parameters, but when evaluated the operand must match a section name, or $MRUN will terminate in error. For example, suppose a metajob contains two essentially different processes, defined in SECTION DIST and SECTION INST. Let &P be a parameter, set to either DIST or INST by the initiator to select the required process. Then the statement:

```
PERFORM &P
```

coded at the very start of the metajob can be used to route control to the required section.

The EXIT statement is coded to return control to the next sequential statement following the last outstanding PERFORM. Just as in Global Cobol, the metajob interpreter, $MRUN, remembers the address of the statement following each PERFORM in an internal stack which is pushed down whenever a PERFORM is interpreted. When an EXIT is processed the top item in the stack is used to determine the statement to which control is to be returned. This stack item is then made available for re-use and the stack contents are 'popped up'. The maximum number of outstanding PERFORMs that can be remembered in this way is 16.

An EXIT statement issued from the very highest level of control, where there is no outstanding PERFORM, generates the dialogue terminator:

```
ENDJOB
```

which, in live mode, causes the entire dialogue so far constructed by the metajob to be actioned. In test mode, where the dialogue has only been printed, the highest level EXIT statement simply causes the test run to terminate.

An EXIT (or JUMP) statement should normally be coded immediately before the ENDMJOB directive, but if you forget to do this the metajob file builder will generate an EXIT statement for you automatically.

It is usual for an EXIT statement to terminate a section physically as well as dynamically, so you will normally code an EXIT instruction immediately prior to the next SECTION statement.

This is not, however, an absolute requirement, and indeed several SECTION statements may appear without intervening EXIT statements to provide, in effect, a subroutine with multiple entry points. For example:

```
SECTION A+B+C
.
.     group A statements
.
SECTION B+C
.
.     group B statements
.
SECTION C
.
.     group C statements
.
EXIT
```

Assume there are no JUMP statements present. Then when SECTION A+B+C is performed, the group A, then the group B and finally the group C statements are executed. Control 'drops through' the statements SECTION B+C and SECTION C, which have no effect.

5.2.3.4 The JUMP Statement

The JUMP statement is equivalent to the Global Cobol GO TO, and can be used to pass control to a section or paragraph name without affecting the stack used by PERFORM and EXIT statements. In structured programming JUMP should be employed sparingly if at all. However, the statement has its uses, particularly in multiple entry point routines. It is coded simply as:

```
JUMP name
```

where name, which may be parameterised, evaluates to a section or paragraph name.

5.2.3.5 Subroutine Example

The following skeleton shows how a metajob performing the two processes DIST and INST might be structured, assuming that these share a common routine SUBR:

```
JOB EXAMP SUBROUTINE SKELETON
     PERFORM &P                    * DIST OR INST
     EXIT                          * END JOB
SECTION DIST
.
     PERFORM SUBR
.
     EXIT                          * RETURN TO CALLER
```

```
SECTION INST
.
     PERFORM SUBR
.
     EXIT                          * RETURN TO CALLER

SECTION SUBR
     EXIT                          * RETURN TO CALLER
ENDMJOB
```

5.2.4 Dialogue Control Statements

The statements described in this section generate no dialogue themselves, but instead control the dialogue resulting from other statements.

5.2.4.1 The SUPPRESS Statement

The SUPPRESS statement should be coded if the dialogue resulting from the metajob is to run in suppressed mode. If this is required a SUPPRESS must be executed before any statement which generates dialogue. This will cause the resulting dialogue to begin with the division header:

```
DIALOGUE DIVISION (SUPPRESS)
```

Normally, if no SUPPRESS is encountered before the first dialogue generating statement, the division header will be simply:

```
DIALOGUE DIVISION
```

Note that if a SUPPRESS statement is met after a statement generating dialogue, $MRUN will terminate in error.

5.2.4.2 The VERIFY Statement

A VERIFY statement can be used at any point in a metajob to set the verification flag on or off. (The flag is off at the very start.) You code either:

```
VERIFY         (to turn the flag on)
```

or:

```
VERIFY OFF          (to turn it off)
```

If the single operand of the instruction is anything apart from the word OFF, the flag will be turned on as though a VERIFY by itself had been coded.

When the verification flag is on, the COPY and MERGE statements (explained later) generate additional dialogue to compare the new files or members copied or merged onto the output volume with the originals on the input volume. You can contrast the expansion of the COPY and MERGE statements in Appendix A4, where verification is on, with those in Appendix A5, where it is off, to see the additional dialogue which results.

5.2.4.3 The INPUT and OUTPUT Statements

The INPUT and OUTPUT statements are used to establish the volume-id and unit-id for the input and output volumes. They are coded:

```
INPUT [volume-id] unit-id
```

or:

```
OUTPUT [volume-id] unit-id
```

For example:

```
INPUT    100
OUTPUT SYSRES $CP
```

The statements generate no dialogue themselves, but cause the interpreter to remember the relevant volume and unit information so that when the next file utility, LIBRARY, or MERGE statement is processed the appropriate units are employed for input and output devices. If a volume-id is supplied, then dialogue to perform volume-id checking will also be generated.

INPUT and OUTPUT statements may be coded at any point within a metajob, apart from within a library group (a set of statements introduced by a LIBRARY statement and terminated by an ENDLIB). They **must** have been used to establish the input and output units before the first file utility or LIBRARY statement is executed, otherwise $MRUN will terminate in error.

5.2.4.4 The COMMAND Statement

The COMMAND statement can be used at any point in a metajob to cause the unit from which $F and $LIB are loaded to be changed from $P (the initial default) to $CP or vice versa. You code either:

```
COMMAND              (to load from $CP)
```

or

```
COMMAND OFF          (to load from $P).
```

Note that this statement only affects loadings of $F and $LIB that are generated automatically by the statements described in sections 5.2.5 and 5.2.6. It does not affect explicit loading of commands by RUN or RESPOND statements.

5.2.5 File Utility Statements

The statements described in this section all generate dialogue for the file utility command, $F. They may be coded anywhere within the metajob apart from within a library group (a set of statements introduced by a LIBRARY statement and terminated by an ENDLIB).

If a file utility statement is encountered but $F is not in control, an <ESCAPE> response to obtain a ready prompt, followed by an *F or $F response to run $F from the program residence or system device (according to whether a COMMAND statement is in force), is generated. Then input and output unit information as established by the INPUT and OUTPUT statements is used to satisfy the input and output device prompts. Next I and O instructions may be employed to check input and output volume-ids. Only then will the dialogue associated with a particular statement, as documented below, be generated. (The preliminary dialogue that may occur is defined in detail in the description of the MOUNT statement in 5.2.5.8 below.)

Note that each file utility statement eventually returns the dialogue to $F's instruction prompt:

```
$66 FILE MAINTENANCE
```

5.2.5.1 The SERIAL Statement

There are three variants of the COPY statement which generate the file utility's SER instruction to establish a product serial number and optional expiry date. You code:

(A) SERIAL

or:

(B) SERIAL number

or:

(C) SERIAL number expiry-date

Variant (A) is used to establish a serial number of 0 with no expiry date, so that template, or free, files may be produced. The dialogue generated is:

```
:SER :<CR>
$66 FILE MAINTENANCE
:
```

Variant (B) is used to establish a product serial number without an expiry date. The dialogue generated is:

```
:SER :number EXP:<CR>
$66 FILE MAINTENANCE
:
```

Variant (C) is used to establish both a product serial number and an expiry date. The dialogue generated is:

```
:SER :number EXP:expiry-date
$66 FILE MAINTENANCE
:
```

5.2.5.2 The PROTECT Statement

The PROTECT statement generates the file utility's FIF statement to fix the serial number and optional expiry date on a single file of the output volume. You code:

PROTECT file-id

and the resulting dialogue is:

```
:FIF :file-id FIXED
$66 FILE MAINTENANCE
:
```

5.2.5.3 The COPY Statement

There are six different variants of the COPY statement which allow you to generate the appropriate dialogue based on the file utility's COP instruction to copy a single file, possibly renaming it or changing its size, or to copy a selection or a group of files. You code:

(A) COPY file-id

or:

(B) COPY file-id new-file-id

or:

(C) COPY file-id new-size

or:

(D) COPY selection-code

or:

(E) COPY file-id <GROUP>

or:

(F) COPY

The first three variants generate dialogue of the form:

```
:COP :file-id TO:new-file-id|CR SIZE:new-size|<CR> COPIED
$66 FILE MAINTENANCE
:
```

This results in the copying of a single file, whose file-id will be changed if variant (B) is used, or whose size will be altered with variant (C). It is not possible to alter both the size and the file-id at the same time, and furthermore the new file-id may not consist entirely of digits, since if it did the statement would be assumed to be variant (C). Similarly, the new file-id must not be the reserved word <GROUP>, since this identifies variant (E).

Variant (D) is used to copy a selection of files identified by means of a selection code which is either:

.   meaning copy all unprefixed files

or:

.xxxxxx   meaning copy all files with suffix xxxxxx

or:

x.   meaning copy all files with prefix x.

The dialogue generated when this variant is used is of the form:

```
:COP :selection-code
first selected file:<CTRL B> COPIED
.
.(other selected files automatically copied)
.
$66 FILE MAINTENANCE
:
```

Variant (E) is used to copy a group of files. The group begins with the file-id specified by the first operand, and terminates when either the end of the volume is reached or a marker file, whose file-id begins with two + signs, is encountered. If the starting file-id is itself a marker it is copied normally and not used to terminate the selection. The dialogue generated is of the form:

```
:COP :<CTRL C> FROM:file-id
.
.(list of copied files)
.
$66 FILE MAINTENANCE
:
```

Variant (F) is used to copy all files of the input volume. The dialogue generated is:

```
:COP :<CTRL B>
.
.(list of copied files)
.
$66 FILE MAINTENANCE
:
```

Note that if the verification flag has been set on by a previous VERIFY instruction, each variant will also generate similar additional dialogue using the file utility's CFI instruction to compare the file, selection or group created on the output volume with the original.

5.2.5.4 The LOAD Statement

The LOAD statement generates the file utility's LOA instruction to load the file utility extension, $FX, which contains the logic for the INS and PAM instructions. You code:

```
LOAD
```

and the resulting dialogue is:

```
:LOA EXTENSION LOADED
$66 FILE MAINTENANCE
:
```

This statement must be issued before an INSTALL or PATCH statement is attempted, otherwise $MRUN will be terminated in error.

Furthermore, you must ensure that the file utility remains loaded from the time the LOAD is executed until the last INSTALL or PATCH requiring the extension takes place. (The LOA instruction requires that $FX is present on the unit from which the file utility itself was loaded.)

5.2.5.5 The INSTALL Statement

The INSTALL statement generates the file utility's INS instruction to install a physical bootstrap on the output volume, using an install file and a bootstrap file from the input volume. You must specify the file-id of the install file and the unit address of the volume upon which it will reside at bootstrap time. You code:

```
INSTALL file-id unit-address
```

and the resulting dialogue is:

```
:INS :file-id SYSINS UNIT:unit-address INSTALLED
$66 FILE MAINTENANCE
:
```

An INSTALL statement should only be executed after a previous LOAD statement has loaded the file utility extension.

5.2.5.6 The PATCH Statement

The PATCH statement generates the file utility's PAM instruction to patch a monitor file on the output volume with the unit address of the system residence device and a code letter (S or T) which indicates whether the monitor is to be used as part of the starter or target bootstrap procedure. The third parameter contains the system stack size and optional minimum memory bank size to use in the form:

```
sssss/bb  (specific sizes)
```
or:
```
sssss          (specific stack size, default bank size)
```
or:
```
0/bb      (default stack size, specific bank size)
```

You code:

```
PATCH monitor-file-id unitx stack/bank
```

(where unitx is the system residence device unit address concatenated with S or T - e.g. 114T) and the resulting dialogue is:

```
:PAM :monitor-file-id SYSRES:unitx STACK/BANKS:stack/bank
$66 FILE MAINTENANCE
```

A patch statement should only be executed after a previous LOAD statement has loaded the file utility extension.

5.2.5.7 The ALLOCATE Statement

The ALLOCATE statement generates the file utility's ALL instruction to create a dummy file on the output volume, with a specified file-id and size. The dummy file has relative sequential organisation, with a record length of one byte. You code:

```
ALLOCATE file-id size
```

and the resulting dialogue is:

```
:ALL :file-id SIZE:size RECORD LENGTH:1 ALLOCATED
$66 FILE MAINTENANCE
:
```

5.2.5.8 The MOUNT Statement

You use the MOUNT statement when you require the file utility to perform volume-id checking on the input or output volume following an INPUT or OUTPUT statement, but do not wish to generate any other file utility instruction. (MOUNT is unnecessary if one of the statements described earlier in this section is to be used, since, as explained in the introduction, each such statement automatically produces the necessary preliminary dialogue if it is required.)

Suppose the file utility to be in control. Then, for example, the statements:

```
INPUT input-volume-id input-unit-id
MOUNT
```

result in the dialogue:

```
$66 FILE MAINTENANCE
:<CR>
$66 INPUT DEVICE:input-unit-id
$66 OUTPUT DEVICE:output-unit-id
$66 FILE MAINTENANCE
:IO :input-volume-id :output-volume-id
$66 FILE MAINTENANCE
:
```

In this, the output-unit-id has been remembered from the last OUTPUT statement. To take a more complicated example, assume the file utility **not** to be in control. Then the statements:

```
INPUT input-unit-id
OUTPUT output-volume-id output-unit-id
MOUNT
```

result in:

```
:<ESCAPE>
GSM READY:*F
$66 INPUT DEVICE:input-unit-id
$66 OUTPUT DEVICE:output-unit-id
$66 FILE MAINTENANCE
:O :output-volume-id
$66 FILE MAINTENANCE
```

```
:
```

You should note that a single INPUT or OUTPUT statement will give rise to at most one I, O or IO instruction for the file utility, since internal input and output volume change flags control the generation. The appropriate flag is set on when the INPUT or OUTPUT statement is processed, and turned off when the associated dialogue is generated.

5.2.5.9 The CHECK Statement
The CHECK statement generates the file utility's VER instruction to verify that every sector on the output volume is readable. You code:

    CHECK

and the resulting dialogue is:

```
:VER volume-id VERIFIED
$66 FILE MAINTENANCE
:
```

5.2.5.10 The DELETE Statement
The DELETE statement generates the file utility's DEL instruction to delete either a single file, or a selection, from the output volume. You code either:

    (A) DELETE file-id
or:
    (B) DELETE selection-code

where in variant (B) the selection code is one of:

    .   meaning delete all unprefixed files
or:
    .xxxxxx   meaning delete all files with suffix xxxxxx
or:
    .x   meaning delete all files with prefix x.

The resulting dialogue from variant (A) is simply:

```
:DEL :file-id DELETED
$66 FILE MAINTENANCE
:
```

and that from variant (B) is:

```
:DEL :selection code<CTRL B>
.
.      (list of selected files, automatically deleted)
.
$66 FILE MAINTENANCE
:
```

5.2.5.11 The INIT Statement
There are two different variants of the INIT statement which can be used to set up an empty volume with a given volume-id. This is accomplished by means of a file utility SCR operation, followed by an optional CHA instruction to change the volume-id, if required. Therefore INIT, despite its name, cannot modify the access option, which is fixed at some previous time when the volume in question was initialised by $V. The two variants of INIT are:

    (A) INIT

(B) INIT volume-id

Variant (A) simply generates a SCR instruction, leaving the existing volume-id unchanged:

```
:SCR existing-volume-id?:Y SCRATCHED
$66 FILE MAINTENANCE
:
```

Variant (B) follows the scratch operation by a CHA instruction to change the volume-id:

```
:SCR existing-volume-id?:Y SCRATCHED
$66 FILE MAINTENANCE
:CHA existing-volume-id TO:volume-id CHANGED
$66 FILE MAINTENANCE
:
```

5.2.6 Librarian Statements

The statements described in this section all generate dialogue for the librarian command, $LIB. The LIBRARY and OLDLIB statements identify a target output library and begin what is known as a **library group**. The group will contain other instructions used to add, replace or delete members of the target library. Eventually it will terminate with the ENDLIB statement, which will close the target library.

The MERGE, TAG, INVOL, STOW and ENDLIB instructions described below can only be coded within a library group. An attempt to execute one of them from outside a group, when there is no outstanding LIBRARY instruction, will lead to $MRUN terminating in error.

You should note that the INPUT and OUTPUT statements, together with the file utility statements, **cannot** be executed within a library group. You will normally, in fact, only use the statements described in this section within the group, together with the structured programming statements. In addition you may code the VERIFY, MESSAGE and RESPOND statements should you so wish.

With the exception of the ENDLIB and EXTRACT statements, each librarian statement leaves the dialogue at the $95 LIBRARY MAINTENANCE prompt. ENDLIB leaves it at the $95 TARGET LIBRARY prompt.

The EXTRACT runs $LIB in a different mode, without an output library, to extract members from input libraries. It cannot be coded within a library group.

5.2.6.1 The LIBRARY Statement

You use the LIBRARY statement to create a new library with a specified file-id on the output volume. This library is initially allocated the maximum amount of contiguous space available, but the space is reduced by a CLOSE TRUNCATE when the dialogue generated by the terminating ENDLIB statement is actioned. An optional second operand of the LIBRARY statement allows you to specify the number of bytes of free space to be left in the library when it is closed, or STUBS to leave spaces for stubs, or MAX to leave the maximum free space. You code:

LIBRARY file-id [size|STUBS|MAX]

The resulting dialogue depends on whether or not the librarian is already in control, and whether or not an OUTPUT statement is

outstanding. In the worst case, when $LIB needs to be loaded from the unit assigned to $P, the dialogue is:

```
:<ESCAPE>
GSM READY:*LIB
$95 TARGET LIBRARY:<CTRL C> VOLUME-ID:output-volume-id
$95 TARGET LIBRARY:file-id UNIT:output-unit-id
$95 NEW?:Y SIZE:<CR>
$95 LIBRARY MAINTENANCE
:I input-volume-id
$95 LIBRARY MAINTENANCE
:
```

The first two lines, which load the librarian, will not be generated if it is already in control, and the third line, which checks the output volume-id, will only be present if an OUTPUT statement is outstanding. The output-unit-id used in the fourth line is that established by the previous OUTPUT statement. The size operand is not employed in this dialogue, but is remembered for use by the ENDLIB statement.

Note that $MRUN will terminate in error if a LIBRARY statement is attempted before both the input and output volumes have been established by INPUT and OUTPUT. An error will also result if two LIBRARY or OLDLIB statements are encountered without an intervening ENDLIB.

5.2.6.2 The OLDLIB statement

You use the OLDLIB statement to update an existing library with a specified file-id on the output volume. An optional second parameter allows you to specify a size which is the number of bytes of free space to be left in the library when it is closed, or STUBS to leave space for stubs. You code:

```
OLDLIB file-id [size|STUBS]
```

The resulting dialogue depends on whether or not the librarian is already in control, and whether or not an OUTPUT statement is outstanding. In the worst case, when $LIB needs to be loaded from the unit assigned to $P, the dialogue is:

```
:<ESCAPE>
GSM READY:*LIB
$95 TARGET LIBRARY:<CTRL C> VOLUME-ID:output-volume-id
$95 TARGET LIBRARY:file-id UNIT:output-unit-id
$95 PROCESS library-title?:Y
$95 LIBRARY MAINTENANCE
:I input-volume-id
$95 LIBRARY MAINTENANCE
:
```

The first two lines, which load the librarian, will not be generated if it is already in control, and the third line, which checks the output volume-id, will only be present if an OUTPUT statement is outstanding. The output-unit-id used in the fourth line is that established by the previous OUTPUT statement. The second operand is not employed in this dialogue, but is remembered for use by the ENDLIB statement.

Note that $MRUN will terminate in error if an OLDLIB statement is attempted before both the input and output volumes have been established by INPUT and OUTPUT. An error will also result if two

LIBRARY or OLDLIB statements are encountered without an intervening ENDLIB.

5.2.6.3 The MERGE statement

There are three variants of the MERGE statement which allow you to generate the appropriate dialogue, based on the librarian's MER or OFF instructions, to merge the contents of a library, or just a single member, into the target library, or introduce stubs for a library of members which will be offline when the target library is resident:

(A) MERGE file-id

or:

(B) MERGE file-id name

or:

(C) MERGE file-id <volume-id>

In variant (A) the MER instruction is generated to include every member of the library whose file-id you supplied:

```
:MER FROM UNIT:input-unit-id
$95 FILE:file-id MEMBER:<CTRL B>
.
. (list of included members)
.
$95 LIBRARY MAINTENANCE
:
```

In variant (B), the MER instruction is used to include just the single member you named in the MERGE statement's second operand. To prevent confusion with variant (C) this name must not begin with a < character and end with a > character. The resulting dialogue is:

```
:MER FROM UNIT:input-unit-id
$95 FILE:file-id MEMBER:name INCLUDED
$95 LIBRARY MAINTENANCE
:
```

In variant (C), if the volume-id is the same as the current output volume, as specified in the last OUTPUT statement, then dialogue as for variant (A) is generated, to merge every member of the library. Otherwise the OFF instruction is generated to indicate that the members currently contained in the program library you have specified will be offline, on the volume whose name you have supplied in angle brackets, when the target library is resident:

```
:OFF UNIT:input-unit-id ON VOLUME:volume-id
$95 FILE:file-id MEMBER:<CTRL B>
.
. (list of stubs introduced)
.
$95 LIBRARY MAINTENANCE
:
```

Alternatively, variant (C) can be used to indicate that a single program, held as an individual file, will be offline, on the volume you have specified, when the target library is resident. The same responses are generated, but the dialogue appears differently, since the file-id specified as the first parameter is not a library-id:

```
:OFF UNIT:input-unit-id ON VOLUME:volume-id
$95 FILE:file-id ON volume-id
$95 LIBRARY MAINTENANCE
:<CTRL B>
$95 LIBRARY MAINTENANCE
```

```
:
```

The input-unit-id provided as a response to the unit prompt in all cases is remembered from the previous INPUT statement. Variants (A) and (B) will generate similar additional dialogue, using the COM instruction in place of MER, if the verification flag has been set on by a previous VERIFY statement.

5.2.6.4 The TAG Statement

The TAG statement is used to replace the first 8 bytes of the target library title with an identifier specified by its first operand. You code:

```
TAG identifier
```

and the resulting dialogue is:

```
:CHA existing library-id NEW LIBRARY-ID:<CR>
$95 OLD TITLE existing-title
$95 NEW TITLE:identifier<CTRL A>
$95 LIBRARY MAINTENANCE
:
```

The existing library title will have been determined from the title of the first library input by a MERGE statement. Trailing spaces within the identifier will be replaced by @-characters in order that the first 8 bytes of the title are overwritten by the dialogue in line 3.

The TAG statement should immediately follow a MERGE statement (although there may be intervening structured programming statements), otherwise $MRUN will be terminated in error.

5.2.6.5 The INVOL Statement

The INVOL statement allows you to change the input volume and unit from within a library group. (Remember you cannot use the INPUT and OUTPUT statements.) You code:

```
INVOL volume-id [unit-id]
```

and the dialogue:

```
:I :volume-id
```

is generated so that $LIB performs volume-id checking on the input volume, and ensures that the target library is still mounted. If you specify a unit-id, that will be used for subsequent input files, otherwise if omitted the current input unit remains in force.

5.2.6.6 The ENDLIB Statement

The ENDLIB statement is used to terminate a library group and close the output library. Following it you may employ another LIBRARY or OLDLIB statement to create an additional output library, or use file utility statements or other statements, such as INPUT or OUTPUT, which are not allowed within a library group. You simply code:

```
ENDLIB
```

resulting in the following dialogue:

```
:TRU NEW SPARE SPACE:size|<CR>|<CTRL B>
$95 LIBRARY MAINTENANCE
:END
```

```
$95 TARGET LIBRARY:
```

If you specified the size in bytes of the spare space required in the LIBRARY or OLDLIB statement which introduced the group, this value will be used in the first line of the dialogue. If you specified STUBS then a reply of <CTRL B> is supplied to leave just enough space to fill the library index with stubs, to a total of 100 members. If you specified MAX then the library will not be truncated. Otherwise, if the library was opened using an OLDLIB statement it will be left at its original size, and if opened by a LIBRARY statement it will be truncated to have no spare space.

5.2.6.7 The EXTRACT Statement

The EXTRACT Statement is used to extract a single named member from a library. You code:

```
EXTRACT Library-id member
```

If the previous instruction was not an extract, then $LIB is loaded using the dialogue:

```
:<ESCAPE>
GSM READY:$LIB
$95 TARGET LIBRARY:<CR>
$95 LIBRARY MAINTENANCE
```

The dialogue to extract the member is:

```
:EXT LIBRARY:library-id UNIT:input-unit
$95 EXTRACT:member TO:<CR> UNIT:output-unit
```

5.2.7 Dialogue Extension Statements

The statements described in this section allow you to generate dialogue for programs other than $F and $LIB.

5.2.7.1 The RUN Statement

When the RUN statement is coded with no operands:

```
RUN
```

an <ESCAPE> response is generated which will normally result in the Global Cobol monitor displaying its ready prompt. Alternatively an operand may be supplied:

```
RUN program-id
```

resulting in the dialogue:

```
:<ESCAPE>
GSM READY:program-id
some prompt:
```

The RUN statement can therefore be used to run any program or command, or if no parameter is specified, to obtain the ready prompt. It should not be coded within a library group, otherwise $MRUN will terminate in error.

5.2.7.2 The RESPOND Statement

The RESPOND statement is used to generate a response, or series of responses, of up to 64 bytes in length. It is coded as:

```
RESPOND response
```

and simply generates the dialogue:

```
:response
```

RESPOND, which may be coded anywhere within a metajob, is usually used in conjunction with RUN to construct dialogue which cannot be generated using the standard metajob statement. For example, the sequence:

```
RUN  $A
RESPOND $PR :&P
```

will, if &P is 500, result in the following dialogue when the metajob containing it is actioned:

```
:<ESCAPE>
GSM READY:$A
$69 UNIT:$PR ADDRESS:500
$69 UNIT:
```

Note that the second and subsequent responses must start with a colon. If a response contains a space, it must be coded as an @ sign, unless it is one of the special control responses, such as <CTRL A>. Thus for example, you might code:

```
RESPOND V6.1@PROGRAM@LIBRARY
```

or:

```
RESPOND DEL :B.<CTRL B>
```

5.2.7.3 The MESSAGE Statement

The MESSAGE statement generates the dialogue to display a message, which appears irrespective of whether or not a SUPPRESS statement has been executed. You code:

```
MESSAGE message
```

resulting in the dialogue:

```
+message+
```

The message may be up to 64 characters long.

5.2.7.4 The PAUSE Statement

The PAUSE statement generates dialogue to display a message and then wait until <CR> is keyed. The message is displayed irrespective of whether or not a SUPPRESS statement has been executed. You code:

```
PAUSE message
```

which results in the dialogue

```
-message-
```

The message may be up to 64 characters long.

5.2.8 Parameter Manipulation Statements

The statements described in this section allow you to set a parameter to a value, and to delete characters from an operand. In particular,

if you have a group of statements which are to be executed several times using different parameters, you can code them as a subroutine, using some spare 'work' parameters, and set up the work parameters before calling the routine.

5.2.8.1 The SET Statement

The SET statement copies the value of the second operand to the parameter specified as the first operand. It is coded as:

```
SET parameter [value]
```

where the first operand must be a single parameter.

For example, if subroutine A runs $A to assign &1 to &2, then you could call it to assign $PR to &P by coding:

```
SET &1 $PR
SET &2 &P
PERFORM A
```

In practice, the subroutine would have to be more complex than this to justify using this technique.

If the second operand is omitted, the parameter is set to null.

5.2.8.2 The MASK Statement

The MASK statement removes all except the bytes specified by a mask from a parameter, resulting in the parameter containing the selected value. It is used to economise on the number of parameters needed, by allowing several short parameters to be combined. It is coded as:

```
MASK parameter mask
```

where the first operand is a single parameter, and the mask is a string of up to 8 characters. Only characters in the parameter which correspond to a 'Y' in the mask are selected.

For example, if &A contains a 3 character unit-id followed by a 3 character unit-address, you could use $A to make the assignment by coding:

```
SET &1 &A
MASK &1 YYY
SET &2 &A
MASK &2 NNNYYY
RUN $A
RESPOND &1 :&2
```

Parameters &1 and &2 are assumed to be available as work parameters.

5.3 The Metajob File Builder ($MJOB)

The $MJOB command is used to create a metajob file from a metajob description and produce a metajob listing. The metajob file is a Global Cobol program file containing the parameter value area, together with encoded metajob statements, which loads at locations #5000 onwards. The metajob listing is simply an annotated print-out of the description, which may contain warning or error messages if the latter is suspect or faulty. If errors are found no metajob file is produced.

5.3.1 The Metajob Description Prompt

MJOB begins by prompting you for the file-id and unit of the metajob description to be processed. If you do not explicitly key a prefix when you input the file-id, $MJOB will assume that the file is conventionally named and will append the S. prefix by default. For example:

```
GSM READY:$MJOB
$700 MJOB DESCRIPTION:EXMJOB UNIT:204
$700 MJOB FILE:
```

indicates that the operator wishes to process metajob description S.EXMJOB on unit 204.

5.3.2 To Quit

To quit and return control to the monitor, simply key <ESCAPE> to any prompt output by $MJOB.

5.3.3 The Metajob File Prompt

When the metajob description has been specified you will be prompted for the file-id and unit-id of the metajob file to be produced. For example:

```
$700 MJOB FILE:EXMTWO UNIT:205
$700 LISTING UNIT:
```

If <CR> is keyed in response to the file prompt the suffix of the metajob description is taken as the file-id of the metajob file,

If <CR> is keyed in response to the unit prompt the unit used is the one specified for the metajob description.

5.3.4 The Listing Unit Prompt

Once metajob file details have been specified the listing unit prompt is displayed:

```
$700 LISTING UNIT:205
$700 MJOB DESCRIPTION NOW BEING PROCESSED
```

If the suffix of the job description is EXMJOB then the above example will cause the listing to be written to file L.EXMJOB on unit 205. If the listing unit is a direct access device, the file will initially be allocated the maximum contiguous space available. Any unused space will be returned once $MJOB completes.

If <CR> is keyed in response to the listing unit prompt, the file will be placed on unit $PR, normally the standard printer.

5.3.5 Processing

Once you have satisfied the listing prompt, the message:

```
$700 DESCRIPTION NOW BEING PROCESSED
```

appears and $MJOB begins to validate the metajob description file and output the metajob file and listing. Once this is finished the following message may appear if $MJOB has detected one or more suspect, though not fatal, conditions:

```
$700 NUMBER OF WARNINGS nnnn
```

where nnnn is the decimal number of warning messages output to the job listing. Finally, one of the following two messages appears:

```
$700 NO ERRORS FOUND - MJOB FILE CREATED
```

or:

```
$700 NUMBER OF ERRORS nnnn - MJOB FILE NOT CREATED
```

and control returns to the monitor, which displays its ready prompt. The first of these indicates that, although warnings may have been flagged, there have been no serious error conditions and a metajob file has been produced. The second message shows that serious errors have occurred and in consequence no metajob file has been created.

5.3.6 Operating Notes

If the listing file is being written to direct access storage and it becomes full, $MJOB displays the message:

```
PRINT FILE EXHAUSTED
```

and control returns to the monitor. If there is not enough spare main memory to create the metajob file, the message:

```
$700 - INSUFFICIENT SPACE TO CREATE MJOB FILE
```

is displayed and control returns to the monitor. All the free space above $MJOB, which occupies approximately 10K bytes, is available for the metajob file, which requires:

385 + S bytes

where S is the size of the statements encoded (see Table 5.2.2). This size is printed at the end of the listing.

If either of these errors occur a metajob file will have been allocated, even though no data will have been output. You can delete the unwanted file either by using $F's DEL instruction, or by rerunning $MJOB.

A metajob file will also have been allocated, and require deletion, if $MJOB is terminated by an irrecoverable I/O error.

5.3.7 Example 1 - Creating Standard Metajob File EXMJOB

The following dialogue takes place when you process metajob description S.EXMJOB to create a conventionally named metajob file, EXMJOB. The job listing is written to the installation's standard printer, $PR. Both the metajob description and the metajob file occupy the volume on unit 204:

```
GSM READY:$MJOB
$700 MJOB DESCRIPTION:EXMJOB UNIT:204
$700 MJOB FILE:<CR> UNIT:<CR>
$700 LISTING UNIT:<CR>
$700 MJOB DESCRIPTION NOW BEING PROCESSED
$700 NO ERRORS FOUND - MJOB FILE CREATED
GSM READY:
```

5.3.8 Example 2 - Testing a Modification to EXMJOB

In this example, having modified S.EXMJOB slightly, we produce a metajob file named EXMTWO on 100 and write the metajob listing, L.EXMTWO, to 101.

```
GSM READY:$MJOB
$700 MJOB DESCRIPTION:EXMJOB UNIT:205
$700 MJOB FILE:EXMTWO UNIT:204
$700 LISTING UNIT:205
$700 MJOB DESCRIPTION NOW BEING PROCESSED
$700 NUMBER OF ERRORS 2 - MJOB FILE NOT CREATED
GSM READY:
```

No metajob file has been output, since the metajob description was faulty.

5.4 Using $MTEST to Initiate a Metajob Run

The $MTEST command is provided to allow you to set up the parameter values of a given metajob file from the console, list these parameters, and then eventually run the metajob. The command therefore serves as a useful test aid. Normally, in live running, you will replace it with your own specialised initiator.

5.4.1 The Mjob File Prompt

$MTEST begins by prompting you for the program-id of the mjob file. If it begins with $ then the file will be loaded from the unit assigned to $CP; otherwise the unit assigned to $P will be used. You can load a metajob file whose name begins with a $-character from the unit assigned to $P by keying an asterisk in place of the initial $. For example:

```
GSM READY:$MTEST
$701 MJOB FILE:*EXAMP
.
.     (parameter listing)
.
$701 PARAMETER SETUP:
```

This dialogue loads the mjob file named $EXAMP from the unit assigned to $P, displays the current parameter values and outputs the parameter setup prompt.

5.4.2 The Parameter Setup Prompt

You may reply to the parameter setup prompt:

$701 PARAMETER SETUP:

with one of the following:

- x=value — where x is a parameter-id, i.e. a digit or upper case letter, and value is zero to eight characters representing the value that the parameter, &x, is to assume. If value is less than eight characters in length, rightmost trailing blanks are inserted in the parameter. In particular, you may key x= to set parameter &x to spaces, the null value;
- LIS — to display the current parameter values on the screen;
- RUN — to run the mjob file once the required parameter values have been established.

The parameter setup prompt is redisplayed following an x=value or LIS response, so that you can continue modifying the parameters. When you reply RUN $MTEST chains to $MRUN which interprets the metajob file.

5.4.3 Operating Notes

$MTEST and $MRUN must be on the same unit. You may, however, key *MTEST in response to the ready prompt to cause $MTEST to be loaded from $P and in this case $MRUN will also be loaded from $P.

Before replying RUN you must ensure that &0 contains one of the following four or five byte character strings:

MJOB in which case $MRUN will create and execute job dialogue;

MJOB1 to execute $MRUN in a test mode in which the metajob instructions executed are listed on file L.$RUN on $PR, the standard printer;

MJOB2 to execute $MRUN in a test mode in which both the metajob instructions executed and the dialogue they would generate are listed on file L.$MRUN on $PR.

Note that MJOB1 and MJOB2 are purely test facilities and do **not** cause real dialogue to be generated and executed.

5.4.4 Example

The test library in Appendix A5 was produced by setting &0 to MJOB2, and &C to X, in the example metajob file, EXMJOB. The following dialogue took place when $MTEST was run to establish these parameters ($MTEST, $MRUN and EXMJOB were individual program files on the unit assigned to $P):

```
GSM READY:*MTEST
$701 MJOB FILE:EXMJOB
.
.       first parameter listing
.
$701 PARAMETER SETUP:0=MJOB2
$701 PARAMETER SETUP:C=X
$701 PARAMETER SETUP:LIS
.
.       second parameter listing
.
$701 PARAMETER SETUP:RUN
(Listing L.$MRUN, showing statements executed and the
dialogue they would generate, is output on $PR)
GSM READY:
```

This parameter listing, automatically output at the beginning of the job, shows the values established by $MJOB: &0 contains MJOB but the other parameters are all spaces (i.e. null).

5.5 Writing Your Own Metajob Initiator

Although the $MTEST program is a useful debugging aid, for live running you will normally provide your own specialised metajob initiator to establish the metajob parameters appropriately. This program must load the metajob file, set up the parameters, and then chain to $MRUN. Appendix A3 contains the compilation listing of a simple initiator developed to parameterise the example metajob.

5.5.1 Loading the Metajob File

Your specialised initiator will know the name of its mjob file, so to load the file all that is necessary is a sequence such as:

```
LOAD metajob-id
ON EXCEPTION STOP RUN
```

The loaded file occupies locations #5000 onwards, so it is Important that your initiator itself does not use these locations. This limits the program size to 20K bytes, or 18.7K if the program is linked to start at the debug address.

5.5.2 Updating the Parameters

Following the LOAD operation the system variable $$EPT points at a 288-byte area containing the 36 metajob parameters, each of which is a PIC X(8) field. They are arranged in the order &0,...&9, &A...&Z. You should define the parameter area by a level 01 group coded in the linkage section, such as:

```
01   PARMS
  02 P-0  PIC X(8)
.
.
.
  02 P-9  PIC X(8)
  02 P-A  PIC X(8)
.
.
.
  02 P-Z  PIC X(8)
```

This area is then made addressable by the statement:

```
BASE PARMS ON $$EPT
```

The parameters can then be accessed individually, the field P-x representing &x. (The & character itself, of course, cannot be used as part of a Global Cobol symbol.) When you set up a parameter value, you must ensure it does not contain leading or embedded spaces, as the first space terminates the parameter's value.

You should note that when a metajob file is created by $MJOB, &0 is set to MJOB and all the other parameters are spaces. However, the parameter block can be updated under the control of a metajob using the STOW instruction, as explained in 5.2.6.5.

You must ensure that &0 contains either MJOB, MJOB1 or MJOB2.

5.5.3 Chaining to $MRUN

Once it has set up the parameters, your initiator should complete by chaining to the metajob interpreter, $MRUN, which will then either create and execute dialogue, or produce a test listing according to the setting of &0. You code either:

```
CHAIN "$MRUN"
```

to execute the command from $CP, or:

```
CHAIN "*MRUN"
```

to run it from $P. The second form will probably be the one most commonly used, since often the initiator, interpreter and the metajob

file itself will be members of a special application program library used for product distribution and installation.

Alternatively, you can chain to $MTEST which will display the parameter values you have set up and display its parameter prompt. You can modify parameters if necessary, and then run the metajob. This is usually the most convenient method of debugging a metajob initiator.

6. Macro Pre-Processor Language

6.1 Introduction

This document describes $MACRO, a general purpose macro processor. The command operates by scanning one or more input text files sequentially and producing a single output text file which normally is used as the source for a subsequent compilation, assembly, or job file build. Because it is normally used before compiling, assembling or jobbing, a program such as $MACRO is known as a **preprocessor**. Its actions are determined by options specified at run time, together with preprocessor language statements which may appear anywhere within the input. These statements are easily recognised because they begin with a special character not employed in the programs $MACRO is used to construct. We have chosen the %-character for this role.

It is important to realise that all $MACRO does is to scan the input files line by line and produce lines of the output file, reporting error and warning conditions on the screen. Obviously, the preprocessor statements play a large role in determining what gets written to the output file. To avoid frequent repetition of the clumsy phrase 'written to the output file' we use the word **generated** instead. Thus the previous sentence becomes: 'Obviously, the preprocessor statements play a large role in determining what gets generated'.

In the description which follows we start by explaining how you run $MACRO assuming that you wish to use the default options appropriate for preprocessing most Global Cobol programs and job files. We then describe the preprocessor language in detail. Section 6.4 defines additional options which can optimise performance or allow you to generate assembler language text. There is an example with listings in Appendix D, and a summary of all the error and warnings messages that can be produced by $MACRO in Appendix E.

Option	Type and Description
JOB MANAGEMENT CONTROL (6.4.1)	
TE/NTE	Terminate job management if errors are detected
TW/**NTW**	Terminate job management if warnings are detected
TRANSLATION (6.4.2)	
*=x	Set comment character. Default is *
. =x	Set label delimiter. Default is .
CONFIGURATION (6.4.3)	
EX=n	Set number of extension variables. Default is 100
MA=n	Set maximum number of macro definitions. Default is 100
NL=n	Set maximum nesting level. Default is 10
TRACE (6.4.4)	
TR=n/**NTR**	Trace preprocessor statements to macro nesting level n
SETUP (6.4.5)	
v=string	Set preprocessor variable v to the specified string

n is an unsigned integer; x is a single character; v is A...Z or 0...99
string may contain embedded blanks. No other part of an option may.
(where options appear as related pairs separated by /, bold text indicates the default)

Table 6.2.3 - Preprocessor Options

6.2 Running $MACRO

6.2.1 The Input File Prompt

When you run $MACRO, it begins by prompting you for the name and unit of the input file. This prompt is repeated until you key <CR> to the input file prompt, or have specified the maximum of 10 input files. For example:

```
GSM READY:$MACRO
$509 INPUT FILE:MACLIB UNIT:204
$509 INPUT FILE:SA100 UNIT:<CR>
$509 INPUT FILE:<CR>
$509 OUTPUT FILE:
```

If you do not supply a file prefix, F. is assumed (standing for first source). When <CR> is keyed to the unit prompt the unit-id last established is used.

The reason $MACRO allows for multiple input files is to enable you to store preprocessor statements, such as macro definitions, to be used in producing a number of different source files in a single initial file known as a **preamble**. Thus, in the example the preamble is the file F.MACLIB and it is to be used to generate information from F.SA100. By changing the third line of the dialogue we could of course use F.MACLIB to generate information using a different file. The point is that if there are changes to the preprocessor statements stored in the preamble you only have to update one file (although when this done you may have to run $MACRO many times to preprocess the changes).

6.2.2 The Output File Prompt

Once all input files have been supplied $MACRO prompts for the name and unit of the output file to be created. For instance:

```
$509 OUTPUT FILE:<CR> UNIT:<CR>
$509 OPTION:
```

If, as in the example, no prefix is keyed, S. is assumed by default. If you key <CR> to the unit prompt the unit-id previously established is used. If you key <CR> to the file prompt the file-id of the last input file is used, but with S. replacing its prefix. For instance, if you followed the dialogue of 6.2.1 with:

```
$509 OUTPUT FILE:<CR>  UNIT:<CR>
$509 OPTION:
```

then the output file S.SA100 would be created on unit 101 from the input files F.MACLIB and F.SA100, which would be scanned in that order.

6.2.3 The Option Prompt

When the output file has been defined $MACRO displays its option prompt. You may reply with any of the option codes defined in Table 6.2.3 (and explained in detail in section 6.4). The prompt reappears to allow you to specify additional options one by one until you key <CR>. For example, to set preprocessor variable %Z to the character string FRED:

```
$509 OPTION:Z=FRED
$509 OPTION:<CR>
$509 PRE-PROCESSING
```

6.2.4 Pre-Processing

Once you key <CR> to the final option prompt $MACRO displays the message:

```
$509 PREPROCESSING
```

and begins generating the output file from the input files. Error and warning messages are reported on the screen and are also written to the output file as comments of the form:

```
*** ERROR explanation
```

or:

```
* WARNING explanation
```

(The asterisk may be replaced by another character using a translation option.) Once preprocessing is complete messages are displayed detailing the number of errors and warnings, and control returns to the monitor. For example, when there are no problems:

```
$509 NUMBER OF ERRORS  0
$509 NUMBER OF WARNINGS  0
$509 PREPROCESSING COMPLETE
GSM READY:
```

6.3 The Preprocessor Language

The description in this chapter assumes that $MACRO's default configuration options are in force.

6.3.1 Macro Definitions

A macro is to a preprocessor what a subroutine is to a high level language. When a named subroutine is called selected statements from it are executed; when a named macro is **referenced**, selected statements from it are generated. Just as the subroutine has to be coded before it is called, so the macro has to be **defined** before it is referenced.

A macro is defined by prefixing the statement lines that make up the **macro body** with the %%MACRO statement, and terminating them with the %%ENDM statement:

```
%%MACRO macroname
     lines
     of
     the
     macro
     body
%%ENDM
```

The %%MACRO statement associates a unique macroname with the definition. The macroname, which may consist of any number of letters and digits, is terminated by a space. Although the name may be more than 8 characters in length $MACRO uses only the first 8 to distinguish between different macros.

A macro definition must appear before any statement which references it. Definitions may not be embedded within definitions, so if two %%MACRO statements are encountered without an intervening %%ENDM an error is returned.

To optimise performance $MACRO holds as many macro definitions as possible in memory, but once the space is exhausted later definitions have to be retrieved from their input file whenever they are referenced. Therefore, if you are using a number of macros, make sure the most frequently used ones appear first in the input.

6.3.2 Macro References

$MACRO considers each input line, apart from a preprocessor statement or comment, to be a potential macro reference of the form:

```
[label.] [macroname string-1 string-2 ...] [* comment]
```

Thus, any characters starting a line which are terminated by a period are considered to be a label and ignored. The next group of characters delimited by spaces are treated as a potential macroname. However, if no macro with this name is defined the line is not a macro reference and it is simply written to the output file and not processed further.

When the line is a macro reference $MACRO splits up the text following the macroname into zero or more different strings, assuming each of them to be separated from its neighbour by one or more spaces. The first string so decoded is placed in macro parameter %1, the second in macro parameter %2, and so on up to %9 if nine such strings are present. The process stops when a comment beginning with an asterisk is encountered, or the line end is met. The statements from the macro definition are then generated. These will normally contain parameter references, %1, %2, etc., so that they can generate different code depending on the macro reference. The situation is analogous to subroutine parameter passing and a simple example should serve to make it clear.

Suppose that a macro ZEROISE is defined as follows:

```
%%MACRO ZEROISE
     MOVE 0 TO %1
%%ENDM
```

Then the following line

```
ZEROISE ALPHA        * CLEAR ALPHA
```

is treated as a macro reference to ZEROISE and generates:

```
MOVE 0 TO ALPHA
```

If a label is used as well:

```
AA100. ZEROISE ALPHA
```

the lines:

```
AA100.
     MOVE 0 TO ALPHA
```

are generated.

If spaces or asterisks are contained in the strings to be passed as parameters, then the strings in question must either begin and end with double quote marks or matching square brackets. The " characters

are passed as part of the parameter whereas the square brackets are not.

For example:

```
ZEROISE "ALPHA BETA"
```

generates the invalid Global Cobol:

```
MOVE 0 TO "ALPHA BETA"
```

whereas:

```
ZEROISE [ALPHA BETA]
```

generates the expected multi-target move:

```
MOVE 0 TO ALPHA BETA
```

Note that this use of square brackets in the preprocessor language itself conflicts with our normal use of them in describing optional constructs. We will take care always to clarify which usage is intended.

In addition to macro parameters %1 to %9, each macro reference has associated with it a special parameter: %0, the **reference counter**. This contains a value which is initially zero, but which is incremented by one every time a macro reference is processed. Since, like the other parameters, it can be concatenated with any text string, %0 allows you to generate unique labels whenever a macro definition is expanded. For example, suppose we have defined ACCUMULATE thus:

```
%%MACRO ACCUMULATE
A%0.
    ADD %1 TO %2
%%ENDM
```

then the two statements:

```
ACCUMULATE ALPHA GAMMA
ACCUMULATE BETA GAMMA
```

would, if they were the first two macro references, generate:

```
A1.
    ADD ALPHA TO GAMMA
A2.
    ADD BETA TO GAMMA
```

Note that just as a subroutine may itself call other subroutines, so the statements within a macro definition may themselves reference other macros. Just as there are always implementation limits on the maximum level of subroutines allowed, so there are limits on how many levels of macro may be nested in this fashion. The default configuration option is to support 10 such levels. Each has its own copy of parameters %0 to %9 just as in some machine languages each subroutine has its own copy of the registers.

6.3.3 Macro Parameters, Preprocessor Variables and Substitution

In addition to the ten parameters employed in macro referencing, there are 126 preprocessor variables named %A to %Z, %%00 to %%99, which can be used anywhere.

Macro parameters and preprocessor variables can each contain strings of between 0 to 31 characters in length. The zero length string is known as SPACE (or SPACES). Initially all preprocessor variables which are not initialised by means of the setup option are set to SPACE. During a macro reference any parameters for which there are no corresponding strings are also set to SPACE. Thus, following:

```
ZEROISE ALPHA
```

the definition is generated with %0 set to the reference count, %1 set to ALPHA and %2 to %9 to SPACE.

In most circumstances, when a parameter or variable is encountered it is replaced by the character string it currently contains, a process referred to as **substitution**. For example, if %%23 contains JIM then a line coded as:

```
* BY %%23MINY %%23, %%23BO IS IN THE %%23NASIUM
```

expands to:

```
* BY JIMMINY JIM, JIMBO IS IN THE JIMNASIUM
```

IF %A is ERO, %B is E and %C is AL then:

```
Z%AIS%B %CPHA
```

expands to:

```
ZEROISE ALPHA
```

These examples show that substitution is applied not only to comment lines but to potential macro reference lines **before** $MACRO checks to see if the macro is defined and sets up the parameters. Indeed, the only places where substitution does not occur are:

- preprocessor comment (%*) and %%PAGE statements;
- the preprocessor variable name identifying the variable to be assigned by a preprocessor function;
- the preprocessor variable or macro parameter tested by the %%IF statement and the condition used by that statement.

In the first case, substitution is avoided so that you can use the % character when writing comments relating to the working of the preprocessor statements themselves. For example:

```
%* IF %1 IS NOT NUMERIC SIGNAL AN ERROR
```

In the second and third cases, assigning and testing parameters or variables, there is no substitution because $MACRO must retain the

name of the parameter or variable in order to carry out the function or test.

Note that the substitution process is limited by the line length. If a line longer than 72 characters results the extra rightmost ones are eliminated and a warning message is generated.

6.3.4 Function Statements

Preprocessor variables can be assigned values using function statements, the general format of which is:

```
function variable-name string
```

There are currently four functions defined: %%STR, %%LEN, %%NUM and %%SPL. The variable-name must be %A to %Z, or %%00 to %%99. Spaces must separate the function, variable-name and string.

Normally the string is considered to begin with its first non-blank character and end with its last non-blank character. If it contains leading or trailing spaces it must be embedded in square brackets. Two adjacent square brackets, [ ], should be coded rather than the characters SPACE to represent the empty string. Any square brackets are eliminated when the variable is set up by the function. The string size after substitution is of course limited by the line length.

6.3.4.1 %%STR - Assign String Value

The %%STR function is used to set a macro variable to the value of a particular string, substitution taking place before the function is evaluated. Thus if %X is 99:

```
%%STR %A -00%X
```

sets %A to the five character string -0099.

If the string contains more than 31 characters only the leftmost 31 will be transferred to the variable.

6.3.4.2 %%LEN - Calculate String Length

The %%LEN function enables you to determine the length of a given string. Thus if %X is 99 as before:

```
%%LEN %A -00%X
```

sets %A to 5 and then:

```
%%LEN %A %A
```

sets %A to 1.

6.3.4.3 %%NUM - Assign Numeric Value

The %%NUM function is used to evaluate a numeric expression, and place the result, an integer in the range -999999999999999 to 999999999999999, in the variable. After substitution the string must be a numeric expression of the form:

```
[ - ] operand [operator operand...]
```

where the square brackets and ellipsis (...) are used as documentation aids to represent optional constructs and repetition respectively, and are not part of the string itself.

An operand may be an unsigned integer in the range 0 to 999999999999999 or another numeric expression enclosed in round brackets. An operator may be one of +, -, / (divide), * (multiply).

All operations are integer, with division truncating fractions so that the results are rounded towards zero. There is no operator precedence; numeric expressions are evaluated strictly from left to right, starting at the lowest level of bracketing.

For example:

```
4 * 3 + 2 = 14
2 + 4 * 3 = 18
2 + (4 * 3) = 14
-(5 +7) = -12
-5 +7 = 2
```

Non-significant leading zeros are suppressed, so with %X = 99 as before:

```
%%NUM %A -00%X
```

sets %A to -99.

6.3.4.4 %%SPL - Split String into Substrings

The %%SPL function allows you to split the string coded as its second operand into two substrings - alpha and beta, for example. Alpha consists of those characters to the left of the first occurrence of the **slicer** string or the entire string if it does not contain the slicer. Beta consists of the characters to the right of the first occurrence of the slicer string, or consists of SPACE if there was no slicer or if it appeared only once as the rightmost character(s) of the original string. If the slicer string itself is SPACE this is treated specially. Alpha becomes the first character of the string and beta the residue. Whatever the slicer is, both substrings will be SPACE if the original string is SPACE.

%%SPL takes the slicer from the preprocessor variable specified as its first operand. It then overwrites this value with substring alpha. Beta is then placed in the first variable's successor. (%B is the **successor** of %A, %C of %B etc.; %%01 is the successor of %Z; %%02 that of %%01; and so on. The last extension variable has no successor and will cause %%SPL to return an error if coded by mistake.)

The following example shows how a string of the form:

```
keyword = value
```

contained in macro parameters %1 is split so that %K contains the keyword and %L the value:

```
%%STR %K =
%%SPL %K %1
```

In this second example macro parameter %2 contains a non-null string which may begin with an initial asterisk. %L is set to the string with

its first character removed. Note how the first line sets the slicer to SPACE by using adjacent square brackets. You cannot use the word SPACE itself, otherwise the variable would be set to those five characters:

```
%%STR %K [ ]
%%SPL %K %2
```

6.3.5 Conditional Preprocessor Statements

To allow statement lines to be generated conditionally %%IF, %%AND, %%OR, %%ELSE and %%END statements are provided to support the familiar structured programming construct:

```
%%IF condition-description
[%%OR statements | %%AND statements]
     statements generated if the condition is true
[%%ELSE
     statements generated if the condition is false]
%%END
```

Here and in the following example the square brackets are used to denote the optional nature of the construct and are not actually coded.

The format of the %%IF statement is:

```
%%IF name [NOT] condition [string]
```

The name is that of a macro parameter or preprocessor variable. Table 6.3.5 lists the conditions you may code. The character and arithmetic comparisons test the value of the parameter or variable supplied as the first operand against that of the string. The type checking comparisons simply check the value of the parameter or variable itself and no string should be provided. NOT if coded as the second operand reverses the meaning of the condition.

When a string is coded it is treated like a function string: it is delimited by its first and last non-blank characters unless it contains leading or trailing spaces in which case it must be enclosed in square brackets. For a numeric comparison the string must assume the form of a numeric expression after substitution. Character comparison proceeds from right to left, byte by byte, using the ASCII collating sequence in exactly the same way that Global Cobol character variables are compared. For numeric Comparison the expression string is evaluated and then the comparison takes place as though two Global Cobol computational quantities were involved. Thus if %A is 34:

```
%A < 345 is true

%A < 40 is true

%A < 4 is true, but ...

%A .< 4 is false
```

Condition	**Description**

TYPE CHECKING	
SPACE or SPACES	True if parameter or variable not defined.
NUMERIC	True if parameter or variable contains an integer in the range: -999999999999999 to 999999999999999
CHARACTER COMPARISON	
= or EQUAL	True if parameter or variable alphabetically equal to string.
> or GREATER	True if parameter or variable alphabetically greater than string.
< or LESS	True if parameter or variable alphabetically less than string.
NUMERICAL COMPARISON	
.= or .EQUAL	True if parameter or variable is arithmetically equal to the string evaluated as a numeric expression.
.> or .GREATER	True if parameter of variable is arithmetically greater than the string evaluated as a numeric expression.
.< or .LESS	True if parameter or variable is arithmetically less than the string evaluated as a numeric expression.

Table 6.3.5 - Conditions Used in %%IF, %%AND & %%OR Statements

These examples show how important it is to use the right type of condition in a test. For instance, if %X is 5 and %Y is 2 the construct:

```
%%IF %X .> 4*%Y
     group-1 statements
%%ELSE
     group-2 statements
%%END
```

causes the group-2 statements to be generated, since the numeric expression evaluates to 8. However, just forget the period in the condition and erroneously code:

```
%%IF %X > 4*%Y
```

and the string involved in this character comparison evaluates to 4*2 and since %X is 5 it is alphabetically greater, so the group 1 statements are generated by mistake.

The %%OR statements and %%AND statements can be used just like the OR and AND statements in Global Cobol to create compound conditions. Their operands have the same format as those of the %%IF statement. As in Global Cobol, more than one %%OR or %%AND statement may be coded following the initial %%IF, but the two types of statement may not be intermingled. That is:

```
%%IF %A = A
%%OR %B = B
%%OR %B SPACES
```

is valid, whereas:

```
%%IF %A = A
%%AND %B = B
%%OR %B SPACES
```

is not.

6.3.6 Error and Fail Handling

If you detect an error during preprocessing you can write an explanatory message to the output file and the screen, and increment the error count maintained by $MACRO, by executing the %%ERROR statement:

```
%%ERROR message
```

which leads to the generation of a message line of the form:

```
**ERROR explanation
```

where the explanation that appears is the substituted message from the %%ERROR statement. $MACRO then continues preprocessing the input file.

If the error you have detected is catastrophic and there is no point in continuing, not even to detect other errors, you can abort $MACRO by coding the %%FAIL statement:

```
%%FAIL message
```

This causes a message line of the form:

```
***FAIL explanation
```

to be written to the output file, which is then closed. The same message is written to the screen and $MACRO aborts the run, returning to the monitor. Job management is terminated, irrespective of the options in force, if it is in control.

6.3.7 The Macro Exit Statement, %%MEXIT

The %%MEXIT statement enables you to exit from a macro definition before the %%ENDM statement is encountered. You use it in conjunction with conditional statements. For instance, suppose macro CALC either takes no parameters, or requires its first parameter to be numeric. You might structure the definition like this:

```
%%MACRO CALC
     %%IF %1 NOT SPACE
          %%IF %1 NOT NUMERIC
               %%ERROR CALC PARAMETER NON-NUMERIC
               %%MEXIT
          %%ELSE
               statements when parameter 1 numeric
          %%END
     %%ELSE
          statements when no parameter
     %%END
%ENDM
```

You can consider the %%MEXIT statement to be the dynamic end of the macro (equivalent, say to a Global Cobol EXIT statement) whilst %%ENDM is the contextual end (somewhat analogous to the ENDPROG statement). You do not, of course, have to code a %%MEXIT immediately before a %%ENDM. $MACRO automatically inserts one if it is needed.

6.3.8 Preprocessor Comments and the %%PAGE Statement

The characters %* appearing together anywhere in a line (even if within square brackets or quotes) are considered to introduce a preprocessor comment. The %* and the characters which follow, up to the line end, are ignored by $MACRO. They are not subject to substitution, they are not written to the output file, and if they appear within a macro definition they are not stored in main memory.

By using %* you can include comments describing how macros or other preprocessor constructs work. This allows you to document your macro definitions without having your comments appearing as the result of every macro reference. If you use language comments for this purpose by mistake they will be generated unnecessarily and substitution will take place, possibly leading to confusing results. For example, contrast this correct way of commenting ACCUMULATE:

```
%%MACRO ACCUMULATE
     %%IF %2 SPACES
          ADD %1 TO SYSACC
          %* UPDATE SYSACC IF %2 OMITTED
     %%ELSE
          ADD %1 TO %2
     %%END
%%ENDM
```

which causes:

```
ACCUMULATE 100
```

to generate:

```
ADD 100 TO SYSACC
```

with what happens if you replace the preprocessor comment with the very similar looking language comment:

```
*UPDATE SYSACC IF %2 OMITTED
```

If this were done:

```
ACCUMULATE 100
```

would generate:

```
ADD 100 TO SYSACC
*UPDATE SYSACC IF OMITTED
```

The comment is not only unnecessary, but parameter substitution, which has eliminated %2 since it is undefined, has rendered it meaningless.

Any line whose first non-blank characters are %%PAGE is ignored by $MACRO, but treated specially by $PRINT. When you use $PRINT to list a file whose file-id begins with an F. prefix ordinary PAGE statements are ignored, but %%PAGE statements cause printing to advance to a new page. Statements are sequenced just as they are when an S. file is printed.

6.4 $MACRO Options

This chapter explains the meaning of the options summarised in Table 6.2.3. The options are strings of characters keyed in response to the prompt:

```
$509 OPTION:
```

In general they may be keyed in any order. The one exception is that option EX, which is used to change the number of extension variables, cannot be employed once an initial value has been established for any variable using an setup option.

6.4.1 Job Management Options

The job management options only take effect when $MACRO is run under job management. Keying TE will cause job management to be terminated if one or more errors are reported. Option TW is the same, but applies to warnings. NTE and NTW specify no termination on errors or warnings respectively. The errors and warnings detected by $MACRO itself are described in Appendix B. In addition, user errors may be reported by means of the %%ERROR preprocessor statement defined in paragraph 6.3.6.

If a job management is to be terminated this happens at the end of the run after the preprocessing complete message appears. The default if neither job management option is supplied is to terminate on errors but not on warnings.

6.4.2 Translation Options

The translation options are provided to allow $MACRO to generate source code for various assembler languages.

By keying <u>*=characte</u>r you can change the start of comment character appropriately. For example, suppose you keyed <u>*=;</u> to preprocess Macro-11. This would cause $MACRO to recognise a semi-colon rather than asterisk as the comment character terminating the strings of a potential macro reference (6.3.2). Fail, error and warning messages would begin:

```
;;; FAIL
;; ERROR
; WARNING
```

respectively, and the trace indicator (6.4.4) would become ;M;. Note however that preprocessor comments would still be considered to start with the %* sequence.

By keying <u>.=character</u> you can alter the label terminator character used to detect initial labels in potential macro reference lines. For example, for Macro-11 source you use .=:. The option <u>.=1</u> is treated specially. It is used for IBM-like assemblers where labels are not terminated specially but any string beginning in column 1 is considered to be a label.

6.4.3 Configuration Options

The configuration options allow you to vary (normally reduce) the amount of memory $MACRO requires for its internal tables. Any space saved will be made available for storing macro definitions, which may increase performance if $MACRO is run in a small user area. Alternatively, for special applications you may find it necessary to increase the deepest nesting level.

The variables %%00 to %%99 are known as **extension variables.** Each one requires 32 bytes of internal table space. Many applications find the 26 variables %A to %Z all they require. If this is the case EX=0 should be keyed to eliminate the extension variable table and save 3200 bytes of memory. In general if only the first n extension variables, %%00 to %%n-1, are used you can key EX=n to save the space associated with the rest of them. Of course, if you have used the EX option but refer to an extension variable you have eliminated, $MACRO generates an error message. The default if EX is not used is that all 100 of them are supported.

By keying MA=number you can define the maximum number of differently named macros you wish to use. This determines the size of the macroname table which requires 11 bytes per entry. The default is 100 macronames, so if you only use 10 keying MA=10 will save 990 bytes. You cannot set MA greater than 100.

The nesting level defines to what extent macro definitions may reference other macros. If no macros are employed the nesting level is zero. If macros are used, but their definitions do not themselves reference other macros, the level is 1. If the definitions do reference other macros, but these so called inner macros do not themselves reference other macros, then the level is 2. If the inner macros reference others with no references the level is 3, ...and so on. Each nesting level requires 320 bytes of table space in which to store its copy of parameters %0 to %9. You can key NL=number to define the number of nesting levels you require. The default is 10, so if you can make do with 1, say, then NL=1 saves 2880 bytes of table space.

6.4.4 The Trace Option

Preprocessor comments, %%PAGE statements and macro definitions are never written to the output file, and when the trace option is **not** specified neither are macro reference lines nor preprocessor statements.

If TR=0 is keyed macro reference lines and preprocessor statements (excluding %%PAGE and %* comments) **outside** macro definitions are generated, but when they are written to the output file they are 'commented out' by inserting the trace indicator *M* at the front of the line or after a label starting a macro reference. For example:

```
AA100. ZEROISE ALPHA
```

is generated as:

```
AA100. *M* ZEROISE ALPHA
```

If %X contains 100:

```
ACCUMULATE %X
```

is generated as:

```
*M* ACCUMULATE 100
```

and:

```
%%NUM %X +1
```

becomes:

    M %%NUM %X 100 +1

As the second and third examples show, the trace always takes place after substitution.

With option TR=0 no statements within macro definitions are traced. If you key TR=1 you will be able to trace statements within macros at nesting level 1 (i.e. the statements generated by macros referenced outside macro definitions) but not the statements generated by inner macros. In general you may key TR=n to trace statements generated at nesting level 0, 1 ... n. If n is greater or equal to the maximum nesting level all possible statements will be traced.

6.4.5 Setup Options

You may use setup options to establish initial values of the preprocessor variables %A to %Z, and %%00 to %%99. You key variable-name = string, leaving off the leading % character(s) and, optionally, leading zeros. The string may contain leading, trailing or embedded blanks; it is terminated by your keying <CR>. For example, to set %C to GOOD SHOW and %%00 to CHAPS with a leading space, key:

    A=GOOD SHOW

and then:

    00= CHAPS

The statement:

    %%STR %X %A%%00

will result in %X being set to GOOD SHOW CHAPS providing, of course, that %A and %%00 have not been changed by earlier preprocessor statements.

Note that if you key any setup options you cannot later use the EX option to change the number of extension variables. If you wish to use EX you should key it before setting up initial values.

Appendix A - Example Metajob System

This simple example metajob is not intended to be realistic, but is provided to show typical expansions of the valid instruction and operand combinations.

Appendix A1 is a listing of the metajob description, source file S.EXMJOB, which is input to $MJOB in order to create the metajob file along with its listing, L.EXMJOB, included in Appendix A2. A simple metajob initiator has been constructed according to the rules laid down in Chapter 5.5 and its compilation listing, L.EXINIT, appears in A3.

Appendix A4 contains a test listing produced from EXMJOB following execution of the initiator. The mode parameter, &O, has been set to MJOB2 so that the statements processed, together with the dialogue they will generate, are printed. The &D, &S, &V and &X parameters used by EXMJOB have been set up so that the optional expiry date operand is supplied for the SERIAL statement, and the optional extension size for the LIBRARY statement. Because &V is non-null a VERIFY statement is issued, causing subsequent COPY and MERGE statements to generate extra dialogue to compare new files and members with the originals. Since &S is non-null, a SUPPRESS statement is executed causing the dialogue to begin:

```
DIALOGUE DIVISION (SUPPRESS)
```

Appendix A5 contains a second test listing from EXMJOB, resulting from a run of $MTEST, as described in 5.4.4. Once again MJOB2 mode has been used so that the executed statements, together with the dialogue they will generate, are printed. This time, however, the &D, &S, &V and &X parameters remain null. This means that the optional expiry date operand for the SERIAL statement, together with the extension size from the LIBRARY statement, are omitted. Since &V is null no VERIFY instruction is issued and COPY and MERGE statements do not generate verification logic. Similarly, because &S is null, no SUPPRESS statement is executed, and the dialogue begins:

```
DIALOGUE DIVISION
```

By referring to either Appendix A4 or A5 you should be able to deduce exactly what dialogue will be generated for any statement you code, and thus supplement the description in Chapter 5.2. You will note that the statements SECTION, IF, ELSE, END and JUMP do not appear on test listings, since they only control the sequence of execution, and neither generate dialogue nor set internal flags. The sequence number on every statement which is printed identifies the originating statement from the metajob description listing in Appendix A2.

```
PRINT OF S.EXMJOB ON UNIT 235 25/02/88 10.20.25 PAGE 1

 1 MJOB EXMJOB EXAMPLE MJOB FILE
 2 *
 3 * RUN THIS METAJOB TWICE
 4 * FIRST TIME ONLY SET &0, &C NON-NULL
 5 * SECOND TIME SET PARAMETERS AS FOLLOWS:
 6 *   &0 MJOB, MJOB1 OR MJOB2
 7 *   &C $  = BOS, ELSE MACHINE ARCHITECTURE CODE
 8 *   &D DATE = EXPIRY DATE
 9 *   &S Y  = SUPPRESS DIALOGUE
 10 *  &V Y  = VERIFY
 11 *  &X 10000 = LIBRARY EXTENSION SIZE
 12 *
 13 SECTION START
 14    IF &S
 15          SUPPRESS
 16    END
 17    MESSAGE EXERCISE HAS STARTED
 18    IF &V
 19          VERIFY
 20          MESSAGE VERIFY
 21    END
 22    INPUT 100
 23    OUTPUT SYSRES 101
 24    SERIAL 12345678 &D
 25    INIT SYSRES
 26    PERFORM SYSRES
 27    PERFORM SYSDEV
 28    SERIAL                  * NO SERIAL NUMBER
 29    INPUT BACRES 100
 30    MOUNT
 31    RUN &*SPECIAL
 32    RESPOND ABD :123 :XYZ@@11 TEXT
 33    RUN
 34    EXIT
```

A1 - MJOB Description, S.EXMJOB - Page 1

```
PRINT OF S.EXMJOB ON UNIT 235  25/02/88 10.20.25       PAGE 2

 35 PAGE "ROUTINE TO CREATE SYSRES"
 36 SECTION SYSRES
 37    LOAD              * $F EXTENSION
 38    INSTALL ++2222A 100     *    SO CAN INSTALL BOOTSTRAP
 39    COPY S.$2222A <GROUP>   * COPY GROUP
 40    COPY $MONITOR
 41    PATCH $MONITOR 100T 2000/40
 42    COPY TESTFILE NEWNAME   * COPY AND RENAME
 43    COPY $.                 * COPY SELECTION
 44    COPY .TEST              * COPY SELECTION
 45    COPY
 46    DELETE TESTFILE
 47    DELETE #.
 48    EXIT
```

A1 - MJOB Description, S.EXMJOB - Page 2

```
PRINT OF S.EXMJOB ON UNIT 235 25/02/88 10.20.25  PAGE 3

 49 PAGE "ROUTINE TO CREATE SYSDEV"
 50 SECTION SYSDEV
 51    OUTPUT SYSDEV 101
 52    INIT
 53    LIBRARY P.DEV &X
 54    MERGE P.ONE
 55    TAG V11.1
 56    INVOL BACIPL 114
 57    MERGE P.TWO
 58    MERGE P.THREE <SYSRES>
 59    STOW MJOBMEMN
 60    PERFORM ETEST
 61    IF &C $
 62    ELSE              * IF &C NOT = $ (BOS)
 63          MERGE P.FOUR $&CTRAM
 64    END
 65    ENDLIB
 66    PROTECT P.DEV
 67    VERIFY OFF
 68    COPY $COMWORK 32766
 69    CHECK
 70    EXIT
 71 *
 72 *
 73 *
 74 SECTION ETEST
 75    IF &C                   * IF CODE DEFINED
 76          EXIT              *     EXIT
 77    END
 78    MESSAGE ARCHITECTURE CODE NOT DEFINED
 79    JUMP ETEST              * ELSE REPEAT MESSAGE ENDLESSLY
 80 ENDMJOB
```

A1 - MJOB Description, S.EXMJOB - Page 3

```
LIST OF      EXMJOB EXAMPLE MJOB FILE      25/07/87 10.35.03 PAGE 1

 1 MJOB EXMJOB EXAMPLE MJOB FILE
 2 *
 3 * RUN THIS METAJOB TWICE
 4 * FIRST TIME ONLY SET &0, &C NON-NULL
 5 * SECOND TIME SET PARAMETERS AS FOLLOWS:
 6 * &0 MJOB, MJOB1 OR MJOB2
 7 * &C $ = BOS, ELSE MACHINE ARCHITECTURE CODE
 8 * &D DATE = EXPIRY DATE
 9 * &S Y = SUPPRESS DIALOGUE
 10 * &V Y = VERIFY
 11 * &X 10000 = LIBRARY EXTENSION SIZE
 12 *
 13 SECTION START
 14    IF &S
 15          SUPPRESS
 16    END
 17    MESSAGE EXERCISE HAS STARTED
 18    IF &V
 19          VERIFY
 20          MESSAGE VERIFY
 21    END
 22    INPUT 100
 23    OUTPUT SYSRES 101
 24    SERIAL 12345678 &D
 25    INIT SYSRES
 26    PERFORM SYSRES
 27    PERFORM SYSDEV
 28    SERIAL                  * NO SERIAL NUMBER
 29    INPUT BACRES 100
 30    MOUNT
 31    RUN &*SPECIAL
 32    RESPOND ABD :123 :XYZ@@11 TEXT
 33    RUN
 34    EXIT
```

A2 - MJOB Description Listing, L.EXMJOB - Page 1

```
LIST OF EXMJOB    ROUTINE TO CREATE SYSRES 25/07/87 10.35.03 PAGE 2

 36 SECTION SYSRES
 37    LOAD              * $F EXTENSION
 38    INSTALL ++2222A 100     * SO CAN INSTALL BOOTSTRAP
 39    COPY S.$2222A <GROUP>   * COPY GROUP
 40    COPY $MONITOR
 41    PATCH $MONITOR 100T 2000/40
 42    COPY TESTFILE NEWNAME   * COPY AND RENAME
 43    COPY $.                 * COPY SELECTION
 44    COPY .TEST        * COPY SELECTION
 45    COPY
 46    DELETE TESTFILE
 47    DELETE #.
 48    EXIT
```

A2 - MJOB Description Listing, L.EXMJOB - Page 2

```
LIST OF EXMJOB    ROUTINE TO CREATE SYSDEV 25/07/87 10.35.03 PAGE 3

 50 SECTION SYSDEV
 51    OUTPUT SYSDEV 101
 52    INIT
 53    LIBRARY P.DEV &X
 54         MERGE P.ONE
 55         TAG V11.1
 56         INVOL BACIPL 114
 57         MERGE P.TWO
 58         MERGE P.THREE <SYSRES>
 59         STOW MJOBMEMN
 60         PERFORM ETEST
 61         IF &C $
 62         ELSE               * IF &C NOT = $ (BOS)
 63          MERGE P.FOUR $&CTRAM
 64         END
 65    ENDLIB
 66    PROTECT P.DEV
 67    VERIFY OFF
 68    COPY $COMWORK 32766
 69    CHECK
 70    EXIT
 71 *
 72 *
 73 *
 74 SECTION ETEST
 75    IF &C                   * IF CODE DEFINED
 76          EXIT              * EXIT
 77    END
 78    MESSAGE ARCHITECTURE CODE NOT DEFINED
 79    JUMP ETEST        * ELSE REPEAT MESSAGE ENDLESSLY
 80 ENDMJOB

METAJOB SIZE 1439 BYTES
```

A2 - MJOB Description Listing, L.EXMJOB - Page 3

```
LISTING OF EXINIT EXAMPLE METAJOB INITIATOR      25/02/88 10.28.53 PAGE 1


 2          PROGRAM EXINIT
 3          *
 4 0000     DATA DIVISION
 5          *
 6 0000     77    ZDAY         PIC 9(6) COMP     * DAY NUMBER
 7 0003     77    ZDATE        PIC 9(6) COMP     * DATE
 8          *
 9          *
 10 0006    LINKAGE SECTION
 11         *
 12 0000    01    PARMS                    * MJOB PARMS
 13 0000     03   P-0          PIC X(8)
 14 0008     03   P-1          PIC X(8)
 15 0010     03   P-2          PIC X(8)
 16 0018     03   P-3          PIC X(8)
 17 0020     03   P-4          PIC X(8)
 18 0028     03   P-5          PIC X(8)
 19 0030     03   P-6          PIC X(8)
 20 0038     03   P-7          PIC X(8)
 21 0040     03   P-8          PIC X(8)
 22 0048     03   P-9          PIC X(8)
 23 0050     03   P-A          PIC X(8)
 24 0058     03   P-B          PIC X(8)
 25 0060     03   P-C          PIC X(8)
 26 0068     03   P-D          PIC X(8)
 27 0070     03   P-E          PIC X(8)
 28 0078     03   P-F          PIC X(8)
 29 0080     03   P-G          PIC X(8)
 30 0088     03   P-H          PIC X(8)
 31 0090     03   P-I          PIC X(8)
 32 0098     03   P-J          PIC X(8)
 33 00A0     03   P-K          PIC X(8)
 34 00A8     03   P-L          PIC X(8)
 35 00B0     03   P-M          PIC X(8)
 36 00B8     03   P-N          PIC X(8)
 37 00C0     03   P-O          PIC X(8)
 38 00C8     03   P-P          PIC X(8)
 39 00D0     03   P-Q          PIC X(8)
 40 00D8     03   P-R          PIC X(8)
 41 00E0     03   P-S          PIC X(8)
 42 00E8     03   P-T          PIC X(8)
 43 00F0     03   P-U          PIC X(8)
 44 00F8     03   P-V          PIC X(8)
 45 0100     03   P-W          PIC X(8)
 46 0108     03   P-X          PIC X(8)
 47 0110     03   P-Y          PIC X(8)
 48 0118     03   P-Z          PIC X(8)
```

A3 - Initiator Compilation Listing, L.EXINIT - Page 1

```
LISTING OF EXINIT EXAMPLE METAJOB INITIATOR 25/02/88 10.28.53 PAGE 2


 50 0008 0 0 PROCEDURE DIVISION
 51          *
 52 0008 0 0 SECTION MAIN
 53          *
 54          * LOAD MJOB FILE AND BASE PARAMETERS
 55          *
 56 000E 0 0      LOAD "EXMJOB"
 57 0020 0 0      ON EXCEPTION STOP RUN
 58 0024 0 0      BASE PARMS ON $$EPT
 59          *
 60          * THE FOLLOWING STATEMENT, WHICH ESTABLISHES TEST MODE 2, MUST
 61          * BE REMOVED IF THIS JOB IS TO GENERATE LIVE DIALOGUE
 62          *
 63 002E 0 0      MOVE "MJOB2" TO P-0
 64          *
 65          * THE ARCHITECTURE CODE IN &C IS TAKEN FROM $$ARCH
 66          *
 67 003C 0 0      MOVE $$ARCH TO P-C
 68          *
 69          * THE EXPIRY DATE IS CURRENT + 100
 70          *
 71 0048 0 0      CALL DT-DY$ USING $$DATE ZDAY
 72 0056 0 0      ADD 100 TO ZDAY
 73 005C 0 0      CALL DY-DT USING ZDAY ZDATE
 74 0068 0 0      CALL DT-DS USING ZDATE P-D
 75          *
 76          * &S AND &V ARE BOTH SET ON, TO SUPPRESS DIALOGUE BUT NOT VERIFICATION
 77          *
 78 0076 0 0      MOVE "Y" TO P-S P-V
 79          *
 80          * &X IS SET TO 2K TO RESERVE THAT NUMBER OF BYTES IN LIBRARY P.DEV
 81          *
 82 0086 0 0      MOVE "2048" TO P-X
 83          *
 84          * FINALLY CHAIN TO $MRUN ON $P
 85          *
 86 0092 0 0      CHAIN "*MRUN"
 87 00A4 0 0 ENDPROG
```

A3 - Initiator Compilation Listing, L.EXINIT - Page 2

```
LISTING OF EXINIT STATISTICS         25/02/88 10.28.53 PAGE 3


NUMBER OF ERRORS 0
NUMBER OF WARNINGS 0


COMPILATION OPTIONS IN FORCE :


TR - TRACE INFORMATION GENERATED IN COMPILATION FILE
NLN - NO LONG NAMES, THE FIRST 6 CHARACTERS ARE SIGNIFICANT
SD - SYMBOLIC DEBUG RECORD GENERATED IN COMPILATION FILE


PROGRAM SIZE = 00A4 BYTES (HEXADECIMAL)


TOTAL NUMBER OF LINES    87 (EXCLUDING COMMENTS 57)


SOURCE FILE -     S.EXINIT ON 235 CREATED 25/02/88
COMPILATION -     C.EXINIT ON 235 SIZE     0,8K
LISTING FILE -    L.EXINIT ON 204 SIZE     3,7K


MACHINE - PERTEC PC-3200
VERSION - V6.0


COMPILATION COMPLETED
```

A3 - Initiator Compilation Listing, L.EXINIT - Page 3

```
TEST OF EXMJOB EXAMPLE MJOB FILE     25/02/88   10.36.07 PAGE     1

NON-NULL PARAMETERS

&0 MJOB2

&C K

&D 12/12/87

&S Y

&V Y

&X 2048




 15 SUPPRESS
 17 MESSAGE
                         DIALOGUE DIVISION (SUPPRESS)
                         +EXERCISE HAS STARTED+
 19 VERIFY
 20 MESSAGE
                         +VERIFY+
 22 INPUT   100
 23 OUTPUT  SYSRES       101
 24 SERIAL  12345678 12/12/87
                         :<ESCAPE> :*F
                         :100 :101
                         :O :SYSRES
                         :SER :12345678 :12/12/87
 25 INIT    SYSRES
                         :SCR :Y
                         :CHA :SYSRES
 26 PERFORM SYSRES
 37 LOAD
                         :LOA
 38 INSTALL ++2222A 100
                         :INS :++2222A    :100
 39 COPY    S.$2222A <GROUP>
                         :COP :<CTRL C> :S.$2222A
                         :CFI :<CTRL C> :S.$2222A
 40 COPY    $MONITOR
                         :COP :$MONITOR :<NULL> SIZE:<NULL>
                         :CFI :$MONITOR WITH:<NULL>
 41 PATCH   $MONITOR 100T
                         :PAM :$MONITOR :100T :2000/40
 42 COPY    TESTFILE NEWNAME
                         :COP :TESTFILE :NEWNAME SIZE:<NULL>
                         :CFI :TESTFILE WITH:NEWNAME
 43 COPY    $.
                         :COP :$. :<CTRL B>
                         :CFI :$. :<CTRL B>
 44 COPY    .TEST
                         :COP :.TEST :<CTRL B>
                         :CFI :.TEST :<CTRL B>
 45 COPY
```

A4 - Test Listing Using EXINIT - Page 1

```
TEST OF EXMJOB EXAMPLE MJOB FILE 25/02/88 10.36.07 PAGE     2

                         :COP :<CTRL B>
                         :CFI :<CTRL B>
 46 DELETE  TESTFILE
                         :DEL :TESTFILE
 47 DELETE  #.
                         :DEL :#.@@@@@@<CTRL B>
 48 EXIT
 27 PERFORM SYSDEV
 51 OUTPUT  SYSDEV       101
 52 INIT
                         :<NULL>
                         :100 :101
                         :O :SYSDEV
                         :SCR :Y
 53 LIBRARY P.DEV  2048
                         :<ESCAPE> :*LIB
                         :<CTRL C> :SYSDEV
                         :P.DEV :101 :Y
                         :<NULL> :<NULL>
 54 MERGE   P.ONE
                         :MER :100 :P.ONE :<CTRL B>
                         :COM :100 :P.ONE :<CTRL B>
 55 TAG     V11.1
                         :CHA :<NULL> :V11.1@@@<CTRL A>
 56 INVOL   BACIPL       114
                         :I :BACIPL
 57 MERGE   P.TWO
                         :MER :114 :P.TWO :<CTRL B>
                         :COM :114 :P.TWO :<CTRL B>
 58 MERGE   P.THREE <SYSRES>
                         :OFF :114 :SYSRES :P.THREE :<CTRL B>
 59 STOW    MJOBMEMN
                         :PAT :MJOBMEMN
                         :MJOB2@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@K@@@@@@@12/1
                         :2/87@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@Y@@@@@@@@@@@@@@@@@@@@@@@Y@@@
                         :@@@@@@@@@@@@2048@@@@@@@@@@@@@@@@@@@@ :<NULL>
 60 PERFORM ETEST
 76 EXIT
 63 MERGE   P.FOUR       $KTRAM
                         :MER :114 :P.FOUR :$KTRAM
                         :COM :114 :P.FOUR :$KTRAM
                         :<NULL>
 65 ENDLIB
                         :TRU :2048 :END
 66 PROTECT P.DEV
                         :<ESCAPE> :*F
                         :114 :101
                         :FIF :P.DEV
 67 VERIFY  OFF
 68 COPY    $COMWORK 32766
                         :COP :$COMWORK :<NULL> SIZE:32766
 69 CHECK
                         :VER
```

A4 - Test Listing Using EXINIT - Page 2

```
TEST OF EXMJOB EXAMPLE MJOB FILE 25/02/88 10.36.07 PAGE      3

 70 EXIT
 28 SERIAL
                          :SER :<NULL>
 29 INPUT   BACRES        100
 30 MOUNT
                          :<NULL>
                          :100 :101
                          :IO :BACRES :SYSDEV
 31 RUN     *SPECIAL
                          :<ESCAPE> :*SPECIAL
 32 RESPOND
                          :ABD :123 :XYZ@@11 TEXT
 33 RUN
                          :<ESCAPE>
 34 EXIT
                          ENDJOB
```

A4 - Test Listing Using EXINIT - Page 3

```
TEST OF EXMJOB EXAMPLE MJOB FILE     25/02/88 10.23.22 PAGE  1

NON-NULL PARAMETERS

&0 MJOB2

&C $




 17 MESSAGE
                         DIALOGUE DIVISION
                         +EXERCISE HAS STARTED+
 22 INPUT   100
 23 OUTPUT  SYSRES       101
 24 SERIAL  12345678
                         :<ESCAPE> :*F
                         :100 :101
                         :O :SYSRES
                         :SER :12345678 :<NULL>
 25 INIT    SYSRES
                         :SCR :Y
                         :CHA :SYSRES
 26 PERFORM SYSRES
 37 LOAD
                         :LOA
 38 INSTALL ++2222A 100
                         :INS :++2222A    :100
 39 COPY    S.$2222A <GROUP>
                         :COP :<CTRL C> :S.$2222A
 40 COPY    $MONITOR
                         :COP :$MONITOR :<NULL> SIZE:<NULL>
 41 PATCH   $MONITOR 100T
                         :PAM :$MONITOR :100T :2000/40
 42 COPY    TESTFILE NEWNAME
                         :COP :TESTFILE :NEWNAME SIZE:<NULL>
 43 COPY    $.
                         :COP :$. :<CTRL B>
 44 COPY    .TEST
                         :COP :.TEST :<CTRL B>
 45 COPY
                         :COP :<CTRL B>
 46 DELETE  TESTFILE
                         :DEL :TESTFILE
 47 DELETE  #.
                         :DEL :#.@@@@@@<CTRL B>
 48 EXIT
 27 PERFORM SYSDEV
 51 OUTPUT  SYSDEV       101
 52 INIT
                         :<NULL>
                         :100 :101
                         :O :SYSDEV
                         :SCR :Y
 53 LIBRARY P.DEV
                         :<ESCAPE> :*LIB
                         :<CTRL C> :SYSDEV
                         :P.DEV :101 :Y
```

A5 - Text Listing USING $MTEST - Page 1

```
TEST OF EXMJOB EXAMPLE MJOB FILE 25/02/88 10.23.22 PAGE 2

                         :<NULL> :<NULL>
 54 MERGE   P.ONE
                         :MER :100 :P.ONE :<CTRL B>
 55 TAG     V11.1
                         :CHA :<NULL> :V11.1@@@<CTRL A>
 56 INVOL   BACIPL       114
                         :I :BACIPL
 57 MERGE   P.TWO
                         :MER :114 :P.TWO :<CTRL B>
 58 MERGE   P.THREE <SYSRES>
                         :OFF :114 :SYSRES :P.THREE :<CTRL B>
 59 STOW    MJOBMEMN
                         :PAT :MJOBMEMN
                         :MJOB2@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@$@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                         :@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ :<NULL>
 60 PERFORM ETEST
 76 EXIT
 65 ENDLIB
                         :TRU :<NULL>     :END
 66 PROTECT P.DEV
                         :<ESCAPE> :*F
                         :114 :101
                         :FIF :P.DEV
 67 VERIFY  OFF
 68 COPY    $COMWORK 32766
                         :COP :$COMWORK :<NULL> SIZE:32766
 69 CHECK
                         :VER
 70 EXIT
 28 SERIAL
                         :SER :<NULL>
 29 INPUT   BACRES       100
 30 MOUNT
                         :<NULL>
                         :100 :101
                         :IO :BACRES :SYSDEV
 31 RUN     *SPECIAL
                         :<ESCAPE> :*SPECIAL
 32 RESPOND
                         :ABD :123 :XYZ@@11 TEXT
 33 RUN
                         :<ESCAPE>
 34 EXIT
                         ENDJOB
```

A5 - Test Listing USING $MTEST - Page 2

Appendix B - Error and Warning Messages From $MJOB

This appendix describes the error and warning messages produced by the metajob file builder, $MJOB, on its listing file. If an **error** occurs, this indicates a serious fault and no metajob file will have been created. $MJOB is able to recover following a **warning**, however, and the description of the warning message specifies the recovery action taken. Metajob files subject to warnings are executable, but it is of course good practice to correct the metajob description so that a clean listing can be obtained.

Usually the error or warning message follows the line to which it applies, but where a missing line is referred to the message is printed where the missing line should have been.

*** WARNING 1 INSTRUCTION NOT RECOGNISED**

The line is not a comment, nor does it identify a directive (MJOB, ENDMJOB, Page), nor is it one of the statements listed in Table 5.2.2. The spurious line is ignored.

***** ERROR 2 ESSENTIAL OPERAND MISSING**

Insufficient operands have been specified for the statement listed, which therefore cannot be compiled. No metajob file will be produced.

*** WARNING 3 SPURIOUS OPERAND IGNORED**

More operands have been coded in the listed statement than are actually required. The extra one(s) are simply ignored.

***** ERROR 4 ILLEGAL PARAMETER IN OPERAND**

An operand contains an & character which is not followed by an asterisk, a digit or an upper case letter. Such a sequence cannot be interpreted, either as an asterisk (in the case of &*) or as a parameter in the range &0...&Z, and in consequence this error occurs.

*** WARNING 5 ENDMJOB DIRECTIVE MISSING**

The end of the metajob description file has been met before encountering an ENDMJOB directive. This warning message is output and then processing continues just as though an ENDMJOB had been met: the metajob description and listing files will be closed, and the metajob file will be created providing no errors have occurred previously.

*** WARNING 6 MJOB DIRECTIVE MISSING**

The first non-comment line of the metajob description should be an MJOB directive. If it is not this warning appears and no title is established for the metajob file.

*** WARNING 7 MJOB DIRECTIVE OUT OF PLACE**

An MJOB directive has been encountered following the first non-current line of the mjob description. The spurious directive is ignored.

***** ERROR 8 SECTION NAME ALREADY DEFINED**

The statement flagged is a SECTION statement whose name operand is the same as that of a SECTION statement previously processed. This error is equivalent to duplicate label definition in a conventional language.

***** ERROR 9 'IF' NEST LEVEL NON ZERO**

The IF nest level was found to be non-zero on encountering a SECTION statement or an ENDMJOB directive, indicating that an END statement is missing.

*** WARNING 10 NO MATCHING 'IF' FOUND**

An END statement has been encountered for which no corresponding IF was previously detected. The spurious END statement will be ignored.

***** ERROR 11 'IF...ELSE...ELSE' DETECTED**

Consecutive ELSEs attached to the same IF statement have been detected. The spurious second and subsequent ELSE statements will be ignored.

***** ERROR 12 MORE THAN 32 NESTED 'IF's**

The maximum nesting level for IF [ELSE] END structures, 32 overall, has been exceeded.

***** ERROR 13 FIRST OPERAND NOT PARAMETER**

The first operand of a SET or MASK statement must consist of a single parameter.

Appendix C - Error Messages From $MRUN

This appendix describes the run-time error messages output by the metajob interpreter, $MRUN, if a condition is detected which prevents it continuing. Each message is displayed on the screen and all (apart from the error 0 message) are printed on the test listing if the interpreter is operating in test mode. All errors are terminal and cause $MRUN to return control to the monitor once the appropriate message has been output.

With exception of error 0, all messages displayed at the screen end with the phrase AT xxxx, where xxxx is the metajob listing line number of the instruction responsible for the error.

***** ERROR 0 NO MJOB FILE SUPPLIED**

This error occurs when $MRUN does not find the four character metajob file identifier, MJOB at location #5000. This may be because the metajob initiator has set up &0 incorrectly; this parameter should be either MJOB, MJOB1 or MJOB2. This error will also take place if you try to execute $MRUN from the ready prompt. The command should be chained to from an initiator which has loaded and parameterised the metajob file.

***** ERROR 1 STATEMENT OUT OF SEQUENCE AT xxxx**

This error means that one of the following sequencing rules have been broken:

- A SUPPRESS statement has been attempted, but dialogue has already been generated by a previous file utility, librarian, or dialogue extension statement.
- An INPUT, OUTPUT, SERIAL, INSTALL, PATCH, COPY, ALLOCATE, MOUNT, RUN or LIBRARY statement has been attempted from within a library group. (A library group is considered to start when a LIBRARY statement has been processed and terminate when an ENDLIB statement is executed.)
- A MERGE, INVOL, STOW or ENDLIB statement has been attempted from outside a library group. There should have been a preceding LIBRARY statement without an intervening ENDLIB.
- A TAG statement does not immediately follow a MERGE (ignoring intervening structured programming statements).

***** ERROR 2 INSTRUCTION NOT RECOGNISED AT xxxx**

The metajob file has been corrupted or the $MJOB command has erroneously generated a statement with an invalid instruction number, which cannot be processed by $MRUN.

***** ERROR 3 SECTION NOT PRESENT AT xxxx**

A PERFORM statement has been attempted, but its operand, which should be a section name, does not correspond to a section name introduced by any SECTION statement appearing within the metajob description.

***** ERROR 4 INPUT/OUTPUT UNIT NOT SET UP AT xxxx**

A LIBRARY or file utility statement has been attempted before input and output device information has been established by the INPUT and OUTPUT statements.

***** ERROR 5 NO DIALOGUE AT xxxx**

An EXIT statement issued from the highest level of control (in order to start execution of the dialogue constructed by $MRUN) has been attempted before any statement that generates dialogue has been processed.

***** ERROR 6 PERFORM FROM 16TH LEVEL AT xxxx**

A PERFORM statement has been issued when there are already 16 PERFORMs outstanding. The maximum subroutine nesting is 16 levels.

***** ERROR 7 NO PREVIOUS LOAD AT xxxx**

An INSTALL or PATCH statement has been attempted but no previous LOAD statement has been executed to bring the file utility extension, $FX, containing the logic of the INS and PAM instructions, into memory. Alternatively a LOAD has been executed but a program other than $F has been run in 1he meantime, so the extension is no longer in memory.

Appendix D - Example $MACRO Listings

The following pages contain printouts of a macro source file F.SAMPLE, and the resulting compiler source file S.SAMPLE. The program is the example program used in the Global Development Cobol User Manual, with some macro statements added to allow the length of the text records to be varied, and to allow all the file creation logic to be omitted.

Two preprocessor variables are used: %L must be set to the length of the text records (1-80), and %D must be set to Y to create a display only version, or N to create the full version. Another variable, %S, is used to hold the size of the file which is calculated as %L*100. The first listing is of the source file F.SAMPLE before it is processed by $MACRO. Note the use of preprocessor comments to explain the parameter usage.

The second printout is the output from $MACRO of a display only version of S.SAMPLE with a text length of 40 characters. The file was produced using the dialogue:

```
GSM READY:$MACRO
$509 INPUT FILE:SAMPLE UNIT:222
$509 INPUT FILE:<CR>
$509 OUTPUT FILE:<CR> UNIT:<CR>
$509 OPTION:D=Y
$509 OPTION:L=40
$509 OPTION:<CR>
$509 PREPROCESSING
$509 NUMBER OF ERRORS 0
$509 NUMBER OF WARNINGS 0
$509 PREPROCESSING COMPLETE
```

Note that if instead the options had been specified as D=N, L=70 the resulting file S.SAMPLE would have been identical to that in the Global Development Cobol User Manual.

```
PRINT OF F.SAMPLE ON UNIT 235 25/02/88 10.36.32 PAGE 1

 1 %* THIS SOURCE CAN BE MACROED TO PRODUCE EITHER THE EXAMPLE PROGRAM
 2 %* SAMPLE BY SPECIFYING %D=N, OR A PROGRAM DISPLAY, WHICH CONTAINS
 3 %* ONLY THE DISPLAY CODE OF SAMPLE, BY SPECIFYING %D=Y
 4 %*
 5 %* THE LENGTH OF THE TEXT RECORDS IS PARAMETER %L, WHICH MUST BE
 6 %* SET TO A VALUE BETWEEN 1 AND 80. THE STANDARD EXAMPLE PROGRAM
 7 %* USES 70.
 8 %*
 9 %%IF %D = Y
 10 PAGE "DISPLAY FILE FROM SAMPLE"
 11 PROGRAM DISPLAY
 12 *
 13 * THIS PROGRAM DISPLAYS THE RECORDS CREATED BY THE EXAMPLE PROGRAM
 14 * SAMPLE
 15 %%ELSE
 16 PAGE "SIMPLE EXAMPLE PROGRAM "
 17 PROGRAM SAMPLE
 18 *
 19 * THIS PROGRAM IS DESIGNED TO SHOW HOW A SIMPLE BOS/COBOL PROGRAM IS
 20 * CONSTRUCTED. IT DEMONSTRATES CONSOLE I/O, SEQUENTIAL AND RANDOM FILE
 21 * ACCESS, I/O EXCEPTION HANDLING AND GENERAL BOS/COBOL STATEMENTS.
 22 *
 23 * THE PROGRAM BUILDS UP A DISK FILE OF %L-BYTE RECORDS WITH
 24 * USER-DEFINED TEXT, AND THEN ALLOWS THE USER TO RETRIEVE AND DISPLAY
 25 * SPECIFIC RECORDS.
 26 %%END
 27 *
 28 DATA DIVISION
 29 *
 30 FD DF ORGANISATION RELATIVE-SEQUENTIAL *FD FOR DISK FILE
 31 ASSIGN TO UNIT "DSK" FILE DFFID
 32 KEY IS DFKEY
 33 RECORD LENGTH IS %L
 34 %%NUM %S %L*100
 35 SIZE IS %S
 36 *
 37 77 C-LINE PIC 9(2) COMP                *LINE COUNT
 38 *
 39 77 Z-REPLY PIC X                       *OPERATOR REPLY
 40 *
 41 77 Z-TEXT PIC X(%L)                    *RECORD TEXT
 42 *
 43 %%IF %D NOT = Y
 44 01 M1                                  *TITLE AND DATE
 45 02 FILLER                              *TITLE
 46    03 FILLER PIC X(10)                 *TITLE INDENT
 47          VALUE SPACES
 48    03 FILLER PIC X(51)                  *TITLE TEXT
 49    VALUE "BOS COBOL DEMONSTRATION PROGRAM"
 50 02 M1-DAT PIC X(8)          *SYSTEM DATE
 51 PAGE
 52 *
 53 * NOTE THAT WHEN A TABLE OF CHARACTER VALUES IS DEFINED AS BELOW,
 54 * THE INDIVIDUAL VALUE STATEMENTS ARE DIRECTLY CONCATENATED.
 55 * THEREFORE THE FULL-LENGTH VALUE MUST BE SUPPLIED FOR EACH TABLE ITEM.
 56 *
 57 01 FILLER
 58 02 K-MES OCCURS 5    PIC X(68)
```

Macro source, F.SAMPLE - Page 1

```
PRINT OF F.SAMPLE ON UNIT 235 25/02/88    10.36.32 PAGE 2

 59    VALUE "     "
 60    VALUE "THIS PROGRAM CREATES A FILE OF USER DEFINED TEXT RECORDS"
 61    VALUE "     "
 62    VALUE "ON UNIT DSK, AND THEN DISPLAYS USER SELECTED RECORDS.     "
 63    VALUE "     "
 64    VALUE "          ----------------             "
 65    VALUE "     "
 66    VALUE "A NULL REPLY IS ALWAYS VALID AND HAS THE EFFECT OF  "
 67    VALUE "     "
 68    VALUE "ESCAPING FROM THE CURRENT LEVEL OF DIALOGUE.        "
 69 *
 70 %%END
 71 PAGE "MAIN CONTROL SECTION"
 72 PROCEDURE DIVISION
 73 *
 74 SECTION AA-MAIN
 75 *
 76 * THIS SECTION MAINTAINS THE FLOW OF CONTROL IN THE PROGRAM
 77 *
 78 *
 79 %%IF %D NOT = Y
 80    MOVE 1 TO C-LINE         *
 81    DO UNTIL C-LINE > 5      *
 82          DISPLAY SPACE                *DISPLAY 5 BLANK LINES
 83          ADD 1 TO C-LINE    *
 84    ENDDO                    *
 85 *
 86    CALL DT-DS$ USING $$DATE M1-DAT *USE DATE SUB-ROUTINE TO
 87                                    *CONVERT SYSTEM DATE TO
 88                                    *SHORT DISPLAY FORMAT
 89 *
 90    DISPLAY M1                      *DISPLAY TITLE AND DATE
 91    DISPLAY SPACE                   *
 92    DISPLAY SPACE                   *AND 3 BLANK LINES
 93    DISPLAY SPACE                   *
 94 *
 95    MOVE 1 TO C-LINE                *
 96    DO UNTIL C-LINE > 5      *
 97          DISPLAY K-MES(C-LINE)        *DISPLAY 5-LINE SIGN-ON MESSAGE
 98          ADD 1 TO C-LINE    *
 99    ENDDO                    *
 100 *
 101 PAGE
 102 FOR TRAINING PURPOSES DELETE THESE THREE ERRONEOUS STATEMENTS.
 103 SEE THE BOS COBOL USER MANUAL (APPENDIX A)
 104 FOR INSTRUCTIONS TO REPLACE THEM WITH CORRECT STATEMENTS.
 105 *
 106   SUBTRACT 16 FROM $$LINE GIVING C-LINE *INITIALISE LINE COUNT
 107 *
 108 AA010.                             *LABEL FOR $DEBUG TRAP
 109 *
 110   DO UNTIL C-LINE NOT POSITIVE       *PRODUCE ENOUGH BLANK LINES
 111         DISPLAY SPACE                *TO POSITION THE
 112         ADD -1 TO C-LINE             *"BOS READY:SAMPLE" DIALOGUE
 113   ENDDO                              *ON THE TOP LINE OF THE SCREEN
 114 *
 115 %%END
 116 AA020.
```

Macro source, F.SAMPLE - Page 2

```
PRINT OF F.SAMPLE ON UNIT 235 25/02/88 10.36.32 PAGE 3

 117                                     *REQUEST AND ACCEPT A FILE
 118                                     *IDENTIFIER FROM THE OPERATOR.
 119                                     *QUIT IF NULL KEYED.
 120   DISPLAY "SPECIFY FILE IDENTIFIER"
 121   ACCEPT DFFID NULL STOP RUN
 122 *
 123 %%IF %D NOT = Y
 124 AA030.
 125                                     *REQUEST AND ACCEPT AN
 126                                     *INSTRUCTION, REPROMPT FOR FILE
 127                                     *IF NULL KEYED.
 128   DISPLAY "CREATE OR DISPLAY(C/D)"
 129   ACCEPT Z-REPLY NULL GO TO AA020
 130 *
 131   IF Z-REPLY EQUAL "C"
 132         PERFORM CA-CREATE           *CREATE FUNCTION IF "C" KEYED
 133         ON EXCEPTION GO TO AA020    *FILE ALREADY EXISTS
 134   ELSE
 135         IF Z-REPLY EQUAL "D"
 136         PERFORM CC-DISPLAY *DISPLAY FUNCTION IF "D" KEYED
 137         ON EXCEPTION GO TO AA020 *FILE DOES NOT EXIST
 138         END
 139   END
 140 *
 141   GO TO AA030                       *REPROMPT FOR INSTRUCTION
 142                                     *IF NOT "C" OR "D" OR WHEN
 143                                     *CREATE/DISPLAY COMPLETE
 144 %%ELSE
 145   PERFORM CC-DISPLAY
 146   ON EXCEPTION GOTO AA020
 147   STOP RUN
 148 %%END
 149 %%IF %D NOT = Y
 150 PAGE "CREATE SECTION"
 151 SECTION CA-CREATE
 152 *
 153 * THIS SECTION OPENS A NEW FILE WITH A RECORD LENGTH OF 70 BYTES ON
 154 * LOGICAL UNIT DSK. THE USER KEYS IN THE TEXT WHICH IS WRITTEN TO
 155 * SEQUENTIAL RECORDS OF THE FILE UNTIL A NULL TEXT STRING IS KEYED,
 156 * WHEN THE FILE IS CLOSE-TRUNCATED (TO RETURN ANY SPARE FILE SPACE)
 157 * AND THE USER INFORMED BY A MESSAGE. CONTROL IS THEN RETURNED TO THE
 158 * "CREATE OR DISPLAY" PROMPT.
 159 *
 160   OPEN NEW DF                       *OPEN A NEW FILE.
 161                                     *IF THE FILE ALREADY EXISTS,
 162                                     *RETURN TO THE FILE IDENTIFIER
 163                                     *PROMPT VIA AN EXCEPTION
 164   ON EXCEPTION
 165         DISPLAY "FILE ALREADY EXISTS"
 166         EXIT WITH 1
 167   END
 168 *
 169 CA010.
 170                                     *REQUEST AND ACCEPT THE TEXT OF
 171                                     *THE NEXT RECORD. IF NULL TEXT,
 172                                     *CLOSE THE FILE
 173   DISPLAY "KEY NEXT TEXT RECORD"
 174   ACCEPT Z-TEXT NEWLINE NULL GO TO CA020
```

Macro source, F.SAMPLE - Page 3

```
PRINT OF F.SAMPLE ON UNIT 235 25/02/88 10.36.32 PAGE 4

 175   WRITE NEXT DF FROM Z-TEXT          *WRITE THE ACCEPTED TEXT TO
 176   ON EXCEPTION                       *THE FILE. IF THE FILE IS FULL,
 177         DISPLAY "FILE FULL"          *CLOSE IT
 178         GO TO CA020
 179   END
 180 *
 181   GO TO CA010                        *ACCEPT NEXT INPUT RECORD
 182 *
 183 CA020.
 184   CLOSE DF TRUNCATE                  *CLOSE THE FILE AND DISPLAY
 185   DISPLAY "FILE "                    *THE FILE-CLOSED MESSAGE
 186   DISPLAY DFFID SAMELINE
 187   DISPLAY " CLOSED" SAMELINE
 188   EXIT                               *RETURN TO MAIN CONTROL
 189 %%END
 190 PAGE "DISPLAY SECTION"
 191 SECTION CC-DISPLAY
 192 *
 193 * THIS SECTION ATTEMPTS TO DO AN "OPEN OLD". FAILURE CAUSES THE
 194 * PROGRAM TO ISSUE A WARNING MESSAGE AND TO THEN RETURN TO THE
 195 * "SPECIFY FILE IDENTIFIER" PROMPT VIA AN EXIT WITH 1. SUCCESS RESULTS
 196 * IN THE USER BEING ASKED FOR A RECORD NUMBER, AND THIS IS USED TO
 197 * RETRIEVE THE SPECIFIED RECORD FROM THE FILE. A NULL RECORD NUMBER
 198 * CAUSES THE FILE TO BE CLOSED AND THE PROGRAM RETURNS CONTROL TO
 199 * THE "CREATE OR DISPLAY" PROMPT. IF THE RECORD NUMBER IS PAST THE
 200 * END OF FILE, AN ERROR MESSAGE IS GIVEN. OTHERWISE THE TEXT OF THE
 201 * RECORD IS DISPLAYED. THE NEXT RECORD NUMBER IS THEN REQUESTED.
 202 *
 203   OPEN OLD DF                        *OPEN THE EXISTING FILE.
 204                                      *IF THE FILE DOES NOT EXIST,
 205                                      *RETURN TO THE FILE IDENTIFIER
 206                                      *PROMPT VIA AN EXCEPTION
 207   ON EXCEPTION
 208         DISPLAY "FILE DOES NOT EXIST"
 209         EXIT WITH 1
 210   END
 211 *
 212 CC010.
 213   DISPLAY "KEY RECORD NUMBER" *REQUEST AND ACCEPT A RECORD
 214   ACCEPT DFKEY NULL GO TO CC020 *NUMBER. IF NULL, CLOSE THE FILE
 215 *
 216   READ DF INTO Z-TEXT                *READ THE SELECTED RECORD
 217                                      *INTO THE BUFFER
 218   ON EXCEPTION                       *IF INVALID RECORD NUMBER
 219         DISPLAY "**ERROR** ATTEMPT TO READ OUTSIDE FILE LIMITS"
 220   ELSE
 221         DISPLAY Z-TEXT               *IF VALID, DISPLAY RECORD
 222   END
 223   GO TO CC010                        *ACCEPT NEXT RECORD NUMBER
 224 *
 225 CC020.
 226   CLOSE DF                           *CLOSE THE FILE AND DISPLAY
 227   DISPLAY "FILE "                    *THE FILE-CLOSED MESSAGE
 228   DISPLAY DFFID SAMELINE
 229   DISPLAY " CLOSED" SAMELINE
 230   EXIT                               *RETURN TO MAIN CONTROL
 231 ENDPROG
```

Macro source, F.SAMPLE - Page 4

```
PRINT OF S.SAMPLE ON UNIT 235  01/03/88 10.36.32       PAGE  1

 1 PAGE "DISPLAY FILE FROM SAMPLE"
 2 PROGRAM DISPLAY
 3 *
 4 * THIS PROGRAM DISPLAYS THE RECORDS CREATED BY THE EXAMPLE PROGRAM
 5 * SAMPLE
 6 *
 7 DATA DIVISION
 8 *
 9 FD DF ORGANISATION RELATIVE-SEQUENTIAL *FD FOR DISK FILE
 10 ASSIGN TO UNIT "DSK" FILE DFFID
 11 KEY IS DFKEY
 12 RECORD LENGTH IS 40
 13 SIZE IS 4000
 14 *
 15 77 C-LINE PIC 9(2) COMP               *LINE COUNT
 16 *
 17 77 Z-REPLY PIC X                      *OPERATOR REPLY
 18 *
 19 77 Z-TEXT PIC X(40)                   *RECORD TEXT
 20 *
```

Compilation Source, S.SAMPLE - Page 1

```
 PRINT OF S.SAMPLE ON UNIT 235       01/03/88 10.36.32      PAGE  2

21 PAGE "MAIN CONTROL SECTION"
22 PROCEDURE DIVISION
23 *
24 SECTION AA-MAIN
25 *
26 * THIS SECTION MAINTAINS THE FLOW OF CONTROL IN THE PROGRAM
27 *
28 *
29 AA020
30                                       *REQUEST AND ACCEPT A FILE
31                                       *IDENTIFIER FROM THE OPERATOR.
32                                       *QUIT IF NULL KEYED.
33    DISPLAY "SPECIFY FILE IDENTIFIER"
34    ACCEPT DFFID NULL STOP RUN
35 *
36    PERFORM CC-DISPLAY
37    ON EXCEPTION GO TO AA020
38    STOP RUN
```

Compilation Source, S.SAMPLE - Page 2

```
PRINT OF S. SAMPLE ON UNIT 222 01/03/88 10.41.35       PAGE  3

 39 PAGE "DISPLAY SECTION"
 40 SECTION CC-DISPLAY
 41 *
 42 * THIS SECTION ATTEMPTS TO DO AN "OPEN OLD". FAILURE CAUSES THE
 43 * PROGRAM TO ISSUE A WARNING MESSAGE AND TO THEN RETURN TO THE
 44 * "SPECIFY FILE IDENTIFIER" PROMPT VIA AN EXIT WITH 1. SUCCESS RESULTS
 45 * IN THE USER BEING ASKED FOR A RECORD NUMBER, AND THIS IS USED TO
 46 * RETRIEVE THE SPECIFIED RECORD FROM THE FILE. A NULL RECORD NUMBER
 47 * CAUSES THE FILE TO BE CLOSED AND THE PROGRAM RETURNS CONTROL TO
 48 * THE "CREATE OR DISPLAY" PROMPT. IF THE RECORD NUMBER IS PAST THE
 49 * END OF FILE, AN ERROR MESSAGE IS GIVEN. OTHERWISE THE TEXT OF THE
 50 * RECORD IS DISPLAYED. THE NEXT RECORD NUMBER IS THEN REQUESTED.
 51 *
 52    OPEN OLD DF                         *OPEN THE EXISTING FILE.
 53                                        *IF THE FILE DOES NOT EXIST,
 54                                        *RETURN TO THE FILE IDENTIFIER
 55                                        *PROMPT VIA AN EXCEPTION
 56    ON EXCEPTION
 57          DISPLAY "FILE DOES NOT EXIST"
 58          EXIT WITH 1
 59    END
 60 *
 61 CC010.
 62    DISPLAY "KEY RECORD NUMBER"  *REQUEST AND ACCEPT A RECORD
 63    ACCEPT DFKEY NULL GO TO CC020      *NUMBER. IF NULL, CLOSE THE FILE
 64 *
 65    READ DF INTO Z-TEXT                 *READ THE SELECTED RECORD
 66                                        *INTO THE BUFFER
 67    ON EXCEPTION                        *IF INVALID RECORD NUMBER
 68          DISPLAY "**ERROR** ATTEMPT TO READ OUTSIDE FILE LIMITS"
 69    ELSE
 70          DISPLAY Z-TEXT                *IF VALID, DISPLAY RECORD
 71    END
 72    GO TO CC010                         *ACCEPT NEXT RECORD NUMBER
 73 *
 74 CC020.
 75    CLOSE DF                            *CLOSE THE FILE AND DISPLAY
 76    DISPLAY "FILE "                     *THE FILE-CLOSED MESSAGE
 77    DISPLAY DFFID SAMELINE
 78    DISPLAY " CLOSED" SAMELINE
 79    EXIT                                *RETURN TO MAIN CONTROL
 80 ENDPROG
```

Compilation Source, S.SAMPLE - Page 3

Appendix E - $MACRO Error and Warning Messages

This appendix describes the error and warning messages generated by $MACRO. Each is written to the output file and displayed on the screen. The initial '*' characters can be replaced by the comment character of your choice using $MACRO's *= option.

In some of the messages, the name of the function appears at the start of the text. This is indicated as %%xyz in the messages below.

**** ERROR - END OF MACRO DEFINITION NOT FOUND**

The end of an input file was reached without encountering a %%ENDM statement to terminate a macro definition.

**** ERROR - 'IF...ELSE...ELSE' DETECTED**
*** WARNING - 'IF' NEST LEVEL NON-ZERO**

At the end of an input file, the IF statement nesting level was non-zero.

**** ERROR - 'IF' NEST LEVEL TOO GREAT**

The maximum nesting level for IF statements is 32.

**** ERROR - INVALID MACRO DEFINITION**

A macro cannot be defined within the definition of a macro.

**** ERROR - LINE OUT OF CONTEXT**

A %%AND or %%OR statement was not preceded by a %%IF statement, or a mixture of %%AND and %%OR statements occurred within one %%IF statement.

*** WARNING - LINE TRUNCATED**

The line has been truncated to 72 characters.

**** ERROR - MACRO NESTING LEVEL TOO DEEP**

The macro nesting level is specified using NL option. The default is 10.

**** ERROR - MISSING MACRO NAME**
**** ERROR - NO MATCHING 'IF' FOUND**

%%END has been found with a preceeding %%IF.

*** WARNING - PARAMETER TRUNCATED**

The parameter value has been truncated to 31 characters.

*** WARNING - TOO MANY PARAMETERS**

A macro can have at most 9 parameters.

**** ERROR - %%xyz INVALID EXPRESSION**

The expression was not correctly formed.

**** ERROR - %%xyz INVALID 'IF' CONDITION**

The conditional test is not one of the valid conditional operators.

**** ERROR - %%xyz INVALID PARAMETER**

A parameter is either a constant when it must be a preprocessor variable, or is not numeric in a numeric conditional test.

**** ERROR - %%xyz MISSING PARAMETER**

A mandatory parameter has been omitted.

**** ERROR - %%xyz VARIABLE OUT OF RANGE**

A preprocessor variable was specified whose number was greater than the number of extension variables specified using the EX=n option.